

Программирование на языке C

Полное введение в язык программирования C

Стефан Кочан

ТРЕТЬЕ ИЗДАНИЕ



Программирование на языке С

Третье издание

Programming in C

Third Edition

Stephen G. Kochan

**DEVELOPER'S
LIBRARY**

Sams Publishing, 201 West 103rd Street, Indianapolis, Indiana 462

Программирование на языке С

Третье издание

Стефан Кочан



Москва • Санкт-Петербург • Киев
2007

ББК 32.973.26-018.2.75

К75

УДК 681.3.07

Издательский дом "Вильямс"

Зав. редакцией *С.Н. Тризуб*

Перевод с английского и редакция *Г.В. Галусева*

По общим вопросам обращайтесь в Издательский дом "Вильямс" по адресу:

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

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

Кочан, Стефан.

К75 Программирование на языке C, 3-е издание. : Пер. с англ. — М. : ООО "И.Д. Вильямс", 2007. — 496 с. : ил. — Парал. тит. англ.

ISBN 5-8459-1088-9 (рус.)

В этой книге раскрыты все возможности языка C, включая последние дополнения, сделанные в стандарте ANSI C99. Процесс обучения построен на примерах завершенных программ, иллюстрирующих каждый новый материал. Вы познакомитесь как с основами языка, так и с хорошей практикой написания программ. Упражнения в конце каждой главы делают книгу идеальным учебным пособием, которым могут пользоваться как учащиеся, так и преподаватели. В приложении кратко изложены все возможности языка и стандартные библиотеки, представленные в удобном для быстрого получения справки виде.

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 фотоконширование и запись на магнитный носитель, если на это нет письменного разрешения издательства Sams Publishing.

Authorized translation from the English language edition published by Sams Publishing, Copyright © 2005

All rights reserved. No part of this book shall be reproduced, stored in a retrieval system, or transmitted by any means, electronic, mechanical, photocopying, recording, or otherwise, without written permission from the publisher. No patent liability is assumed with respect to the use of the information contained herein. Although every precaution has been taken in the preparation of this book, the publisher and author assume no responsibility for errors or omissions. Nor is any liability assumed for damages resulting from the use of the information contained herein.

Russian language edition published by Williams Publishing House according to the Agreement with R&I Enterprises International, Copyright © 2006

ISBN 5-8459-1088-9 (рус.)

ISBN 0-672-32666-3 (англ.)

© Издательский дом "Вильямс", 2007

© by Sams Publishing, 2005

Оглавление

Предисловие	17
Об авторе	18
Благодарности	19
Глава 1. Введение	21
Глава 2. Несколько основных принципов	25
Глава 3. Компиляция и запуск первой программы	31
Глава 4. Переменные, типы данных и арифметические выражения	41
Глава 5. Программные циклы	61
Глава 6. Принятие решений	81
Глава 7. Массивы	109
Глава 8. Функции	131
Глава 9. Структуры	171
Глава 10. Символьные строки	197
Глава 11. Указатели	231
Глава 12. Операции с битами	271
Глава 13. Препроцессор	291
Глава 14. Еще о типах данных	313

Глава 15. Работа с большими программами	323
Глава 16. Ввод и вывод в языке C	335
Глава 17. Дополнительные возможности	361
Глава 18. Отладка программ	377
Глава 19. Объектно-ориентированное программирование	397
Приложение А. Справочник по языку C	409
Приложение Б. Стандартные библиотеки языка C	451
Приложение В. Компиляция программ с помощью gcc	475
Приложение Г. Часто встречающиеся ошибки	479
Приложение Д. Ресурсы	483
Предметный указатель	487

Содержание

Предисловие	17
Об авторе	18
Благодарности	19
Глава 1. Введение	21
Глава 2. Несколько основных принципов	25
Программирование	25
Языки высокого уровня	26
Операционные системы	26
Компиляция программ	27
Интегрированная среда разработки	30
Интерпретаторы	30
Глава 3. Компиляция и запуск первой программы	31
Компиляция	31
Запуск программы	32
Рассмотрим первую программу	33
Отображение значений переменных	35
Комментарии	37
Упражнения	38
Глава 4. Переменные, типы данных и арифметические выражения	41
Работа с переменными	41
Типы данных и константы	43
Базовый тип int	43
Хранение и диапазоны	44
Вещественный тип float	44
Числа удвоенной точности double	45

Символьный тип <code>char</code>	45
Булев тип <code>_Bool</code>	46
Спецификаторы типов: <code>long</code> , <code>long long</code> , <code>short</code> , <code>unsigned</code> , <code>signed</code>	47
Арифметические выражения	49
Целочисленная арифметика и оператор унарный минус	52
Деление по модулю	54
Целочисленные и вещественные преобразования	56
Оператор приведения типов	57
Объединение операций с присваиванием. Операторы присваивания	58
Типы <code>_Complex</code> и <code>_Imaginary</code>	59
Упражнения	59
 Глава 5. Программные циклы	 61
Цикл <code>for</code>	62
Операторы отношения	63
Выравнивание вывода	67
Ввод данных	68
Вложение циклов <code>for</code>	70
Варианты записи цикла <code>for</code>	71
Несколько выражений	71
Пропуск полей	72
Объявления переменных	72
Цикл <code>while</code>	72
Цикл <code>do</code>	76
Утверждение <code>break</code>	78
Утверждение <code>continue</code>	78
Упражнения	78
 Глава 6. Принятие решений	 81
Оператор <code>if</code>	81
Оператор <code>if-else</code>	84
Составные операторы отношения	87
Вложенные операторы <code>if</code>	89
Конструкция <code>else if</code>	91
Оператор <code>switch</code>	97
Булевы переменные	100
Оператор условия	104
Упражнения	106
 Глава 7. Массивы	 109
Объявление массива	110
Использование элементов массива в качестве счетчиков	113
Генерация чисел Фибоначчи	116
Использование массивов для генерации простых чисел	117

Инициализация массивов	118
Символьные массивы	120
Преобразования системы счисления с помощью массивов	121
Спецификатор const	123
Многомерные массивы	125
Массивы с переменной длиной	127
Упражнения	129
Глава 8. Функции	131
Объявление функций	131
Аргументы и локальные переменные	134
Объявление прототипа функции	135
Возврат результатов работы функции	137
Функции вызывают функции	141
Объявление возвращаемых типов и типов параметров	144
Проверка аргументов	145
Нисходящее программирование	147
Функции и массивы	147
Оператор присваивания	151
Сортировка элементов массива	152
Многомерные массивы	155
Многомерный массив переменной длины и функции	157
Глобальные переменные	159
Автоматические и статические переменные	163
Рекурсивные функции	165
Упражнения	168
Глава 9. Структуры	171
Структура для хранения даты	171
Использование структур в выражениях	173
Функции и структуры	176
Структура для хранения значений времени	181
Инициализация структур	183
Составные литералы	184
Массивы структур	185
Структуры, содержащие структуры	187
Структуры, содержащие массивы	189
Варианты структур	192
Упражнения	193

Глава 10. Символьные строки	197
Символьные массивы	197
Символьные строки переменной длины	200
Инициализация и отображение символьных массивов	202
Сравнение двух символьных строк	204
Ввод символьных строк	206
Ввод отдельных символов	208
Строка Null	212
Переходные символы	214
Более подробно о строковых константах	217
Символьные строки, структуры и массивы	218
Улучшенный метод поиска	220
Операции с символами	224
Упражнения	227
 Глава 11. Указатели	 231
Объявление указателей	231
Использование указателей в выражениях	235
Указатели на структуры	236
Указатели в структурах	238
Связанные списки	240
Ключевое слово const и указатели	247
Указатели и функции	248
Указатели и массивы	253
Оптимизация программ	256
Это массив или это указатель?	257
Указатели на символьные строки	258
Строковые символьные константы и указатели	260
Еще об операторах инкремента и декремента	261
Операции с указателями	264
Указатели на функции	265
Указатели и адреса памяти	267
Упражнения	268
 Глава 12. Операции с битами	 271
Операции с битами	272
Поразрядный оператор &	273
Поразрядный оператор	275
Поразрядный оператор ^	276
Поразрядный оператор ~	277
Оператор <<	278
Оператор >>	279

Функция сдвига	280
Ротация битов	281
Битовые поля	284
Упражнения	288
Глава 13. Препроцессор	291
Утверждение <code>#define</code>	291
Модифицируемые программы	295
Переносимые программы	296
Более развитые возможности	297
Аргументы и макроопределения	299
Переменное число аргументов в макроопределении	302
Оператор <code>#</code>	302
Оператор <code>##</code>	303
Утверждение <code>#include</code>	304
Системные подключаемые файлы	306
Условная компиляция	307
Утверждения <code>#ifdef</code> , <code>#endif</code> , <code>#else</code> и <code>#ifndef</code>	307
Множественное включение подключаемых файлов	308
Утверждения препроцессора <code>#if</code> и <code>#elif</code>	309
Утверждение <code>#undef</code>	310
Упражнения	310
Глава 14. Еще о типах данных	313
Перечислимые типы данных	313
Утверждение <code>typedef</code>	316
Приведение типов	319
Знаковое расширение	320
Приведение аргументов	321
Упражнения	322
Глава 15. Работа с большими программами	323
Разделение программы на несколько файлов	323
Компиляция нескольких исходных файлов	
с помощью командной строки	324
Связь между модулями	326
Эффективное использование заголовочных файлов	330
Утилиты для работы с большими программами	331
Утилита <code>make</code>	332
Утилита <code>cvs</code>	333
Утилиты Unix: <code>ar</code> , <code>grep</code> , <code>sed</code> и т.д.	334

Глава 16. Ввод и вывод в языке C	335
Ввод и вывод символов: getchar и putchar	336
Форматированный ввод-вывод: printf и scanf	336
Функция printf	336
Функция scanf	343
Операции ввода-вывода для файлов	347
Перенаправление ввода-вывода в файл	347
Конец файла	349
Функции для работы с файлами	350
Функция fopen	350
Функции getc и putc	352
Функция fclose	352
Функция feof	354
Функции fprintf и fscanf	355
Функции fgets и fputs	355
Функции stdin, stdout и stderr	356
Функция exit	357
Переименование и перемещение файлов	358
Упражнения	359
 Глава 17. Дополнительные возможности	 361
Еще два утверждения	361
Утверждение goto	361
Утверждение null	362
Объединения	363
Оператор “запятая”	365
Квалификаторы	366
Квалификатор register	366
Квалификатор volatile	367
Квалификатор restrict	367
Аргументы командной строки	367
Динамическое распределение памяти	371
Функции calloc и malloc	372
Оператор sizeof	372
Функция free	375
 Глава 18. Отладка программ	 377
Отладка с помощью препроцессора	377
Отладчик gdb	382
Работа с переменными	385
Отображение исходного файла	386
Контроль выполнения программы	387
Вставка точек останова	387
Пошаговое выполнение	388

Список и удаление точек останова	391
Трассировка стека	391
Вызов функций и установка массивов и структур	392
Получение справки с помощью gdb	393
Дополнительные возможности	394
Глава 19. Объектно-ориентированное программирование	397
Что такое объект?	397
Экземпляры и методы	398
Программы для работы с дробями	400
Определение класса для работы с дробями на языке Objective-C	400
Определение класса на языке C++ для работы с дробями	404
Определение класса на языке C# для работы с дробями	406
Приложение А. Справочник по языку C	409
1.0. Диграфы и идентификаторы	409
1.1. Символьные диграфы	409
1.2 Идентификаторы	409
2.0. Комментарии	410
3.0. Константы	411
3.1. Целочисленные константы	411
3.2. Вещественные константы	412
3.3. Символьные константы	412
3.4. Строки символов	413
3.5. Перечислимые константы	414
4.0. Типы данных и их объявления	414
4.1 Объявления	414
4.2. Стандартные типы данных	414
4.3. Производные типы данных	416
4.4. Перечислимые типы данных	422
4.5. Утверждение typedef	422
4.6 Модификаторы типа const, volatile и restrict	423
5.0. Выражения	423
5.2. Операторы языка C	424
5.2. Константные выражения	427
5.3. Арифметические операторы	428
5.4. Логические операторы	428
5.5 Операторы отношения	429
5.6. Поразрядные операторы	429
5.7. Операторы инкремента и декремента	429
5.8. Операторы присваивания	430
5.9. Условные операторы	430
5.10. Оператор приведения типа	430
5.11. Оператор sizeof	430
5.12. Оператор запятая	431

5.13. Основные операции с массивами	431
5.14. Основные операции со структурами	431
5.15. Основные операции с указателями	432
5.16. Составные литералы	434
5.17. Преобразования базовых типов данных	435
6.0. Классы памяти и область видимости	436
6.1. Функции	436
6.2. Переменные	436
7.0. Функции	438
7.1. Объявление функций	438
7.2. Вызов функции	439
7.3. Указатели на функции	439
8.0. Утверждения	440
8.1. Составные утверждения	440
8.2. Утверждение break	440
8.3. Утверждение continue	440
8.4. Цикл do	440
8.5. Цикл for	441
8.6. Утверждение goto	441
8.7. Условие if	441
8.8. Утверждение null	442
8.9. Утверждение return	442
8.10. Утверждение switch	443
8.11. Цикл while	444
9.0. Препроцессор	444
9.1. Триграфные последовательности	444
9.2. Директивы препроцессора	445
9.3. Предопределенные идентификаторы	450

Приложение Б. Стандартные библиотеки языка C	451
Стандартные заголовочные файлы	451
stddef.h	451
limits.h	452
stdbool.h	453
float.h	453
stdint.h	453
Функции для работы со строками	454
Работа с памятью	456
Функции для работы с символами	457
Функции ввода-вывода	457
Функции преобразования форматов в памяти	462
Преобразование строка-значение	463

Функции динамического размещения памяти	465
Математические функции	465
Арифметика комплексных чисел	471
Функции общего назначения	473
Приложение В. Компиляция программ с помощью gcc	475
Формат команд	475
Опции командной строки	476
Приложение Г. Часто встречающиеся ошибки	479
Приложение Д. Ресурсы	483
Ответы на вопросы, список опечаток и т.д.	483
Язык программирования C	483
Книги	483
Web-сайты	484
Группы новостей	484
Компиляторы и интегрированные среды разработки	484
Разное	485
Объектно-ориентированное программирование	485
Язык C++	485
Язык C#	485
Язык Objective-C	486
Инструментарий разработчика	486
Предметный указатель	487

Предисловие

Трудно поверить, что уже прошло 20 лет с тех пор, как я написал книгу *Программирование на языке C*. Тогда только еще одна книга по языку C была в продаже, это *Программирование на языке C* авторов Брайана Кернигана и Денниса Ричи. Как изменились времена!

Когда в начале 1980-х годов появился стандарт ANSI C, эта книга была разделена на две книги: оригинальное издание по-прежнему называлось *Программирование на языке C*, а вторая книга была названа *Программирование на ANSI C*, в которой описывался стандарт ANSI C. Это было обусловлено тем, что для разработчиков компиляторов обычно требуется несколько лет, чтобы выпустить компиляторы на рынок и сделать их вездесущими. Я понимал, что будет невозможно представить и ANSI, и не-ANSI C в одном и том же тексте обучающей программы, и это стало причиной разделения книги.

Стандарт ANSI C изменялся несколько раз с тех пор, как первый вариант был издан в 1989 году. Последняя версия, названная C99, послужила главной причиной издания этой книги, в которой сделан акцент на тех изменениях языка C, что обусловлены новым стандартом.

В дополнение к описанию особенностей стандарта C99, эта книга также включает две новые главы. В первой обсуждается отладка программ на языке C. Во второй предлагается краткий обзор получившей широкое распространение технологии объектно-ориентированного программирования, или ООП. Эта глава была добавлена потому, что несколько популярных языков ООП базируются на C. Это C++, C#, Java и Objective-C.

Я искренне благодарен тем, кто пользовался моей книгой в течение нескольких лет. Письма, которые я получал, приносили мне большое удовлетворение. И это остается моим главным побуждением для того, чтобы продолжать писать. Всех, открывающих книгу впервые, я приветствую и надеюсь, что эта книга удовлетворит все ваши ожидания.

Стефан Кочан

Июнь 2004

steve@kochan-wood.com

Об авторе

Стефан Кочан разрабатывал программное обеспечение на языке программирования C в течение более чем 20 лет. Он автор и соавтор нескольких пользующихся широким спросом книг по языку C, включая *Программирование на языке C*, *Программирование в ANSI C* и *Трудности в программировании на C*, а также нескольких книг по операционной системе Unix, включая *Исследование Системы Unix*, *Программирование командного процессора Unix* и *Безопасность систем Unix*. Самое последнее издание Кочана, *Программирование на Objective-C*, является учебником по объектно-ориентированному языку программирования, который базируется на C.

Благодарности

Я хочу поблагодарить ряд людей за помощь в подготовке различных версий данной книги: это Дуглас Маккормик, Джим Шарф, Генри Табикман, Дик Фритз, Стив Левай, Тони Ианайнно и Кен Браун. Я также хочу поблагодарить Генри Маллиша из Нью-Йоркского университета за его советы при написании книги и за помощь в ее издании.

В издательстве Sams я хотел бы поблагодарить выпускающего редактора Марка Ренфроу и редактора проекта Дэна Нотта. Также хочу выразить благодарность редактору по копированию Карен Аннетт и техническому редактору Брайдену Джоунсу. Наконец, я хотел бы поблагодарить всех других людей в издательстве Sams, которые были вовлечены в этот проект, хотя я непосредственно с ними и не работал.

От издательства

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

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

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

E-mail: info@williamspublishing.com

WWW: <http://www.williamspublishing.com>

Адреса для писем из:

России: 115419, Москва, а/я 783

Украины: 03150, Киев, а/я 152

Введение

Язык программирования С был создан Деннисом Ричи в лаборатории фирмы Bell в начале 1970-х годов. Однако только через десять лет этот язык стал широко распространен и популярен. Задержка произошла потому, что в то время еще не было удачных компиляторов языка С, доступных для коммерческого использования вне лабораторий фирмы Bell. Росту популярности языка С способствовала и популярность операционной системы Unix. Эта операционная система также разрабатывалась в лабораториях Bell, а язык программирования С использовался для написания этой системы.

Фактически подавляющее большинство операционных систем (более чем 90 %) были написаны на языке С! Огромный успех первых персональных компьютеров IBM-PC и его клонов скоро сделал MS DOS самой популярной средой программирования для языка С. С ростом популярности языка С в различных операционных системах, увеличивалось количество разработчиков и продавцов компиляторов С. В основном эти версии языка С базировались на приложении, созданном при издании первой книги по языку программирования С, *Программирование на языке С*, написанной Брайаном Керниганом и Деннисом Ричи. К сожалению, это приложение не обеспечивало полного и однозначного определения языка С, что заставляло разработчиков компиляторов интерпретировать некоторые аспекты языка самостоятельно.

В начале 1980-х годов назрела необходимость стандартизации языка С. В Американском национальном институте стандартов (ANSI), организации, которая работает с такими проблемами, в 1983 году был сформирован комитет ANSI C (названный X3J11), чтобы стандартизировать данный язык. В 1989 году работа комитета была ратифицирована, а в 1990 году был издан первый официальный документ по стандарту языка С.

Поскольку С используется во всем мире, Международная организация по стандартизации (ISO) также была привлечена к работе по его стандартизации. Был принят стандарт, который называли ISO/IEC 9899:1990. Впоследствии были сделаны дополнительные изменения в язык С. Самый последний стандарт был принят в 1999 году. Этот стандарт известен как ANSI C99, или ISO/IEC 9899:1999. Именно эта версия языка рассматривается в данной книге.

Язык С является языком высокого уровня, но в нем все же заложены возможности, которые позволяют пользователю работать непосредственно с аппаратными средствами ЭВМ и общаться с компьютером на достаточно низком уровне. И хотя С — структурированный язык программирования общего назначения, эти возможности были изначально заложены в этот язык, что позволяет непосредственно работать с памятью и предоставляет программисту мощные средства и гибкость.

Эта книга научит вас программировать на языке C. Предполагается, что у читателя нет никаких предварительных знаний по языку. В целом, книга написана так, чтобы к ней мог обратиться как новичок, так и опытный программист. Если у вас уже есть опыт программирования, то вы сможете убедиться, что язык программирования C имеет уникальные возможности, которыми не обладают другие языки программирования. В этой книге обстоятельно рассматривается каждая особенность языка. Для демонстрации возможностей языка представлены небольшие примеры программ.

Рекомендуем опробовать каждую программу, представленную в этой книге, и сравнить результаты, полученные на вашей системе, с приведенными в тексте. Это позволит вам не только изучить язык и его синтаксис, но и ознакомиться со всем процессом обработки исходных программ на языке C и получения исполняемого файла. Вы должны обратить внимание на читабельность программы, о чем неоднократно упоминается в тексте книги. Программы должны быть написаны так, чтобы они могли легко читаться и самим разработчиком, и теми, кто будет сопровождать эти программы. Если вы начинающий программист, то довольно скоро увидите, что хорошо структурированные программы почти всегда более легки в написании, их проще отлаживать и изменять. Кроме того, разрабатывая удобочитаемые программы, вы сможете выработать естественный стиль написания структурированных программ.

Поскольку эта книга была написана как учебник, то тема, обсуждаемая в очередной главе, базируется на предварительно изученном материале. Поэтому максимальная польза при чтении этой книги будет получена только при последовательном изучении каждой главы. Книга не годится для поверхностного обзора. Вы должны также выполнять упражнения, которые представлены в конце каждой главы, прежде чем перейти к следующей главе.

В главе 2, “Несколько основных принципов”, описана фундаментальная терминология языков высокого уровня и процесс компиляции программ. Эта глава поможет читателю вспомнить или изучить основные понятия, используемые в данной книге.

Начиная с главы 3, “Компиляция и запуск программы”, вы будете постепенно осваивать язык программирования C.

Дойдя до главы 16, “Операции ввода и вывода в C”, вы уже будете знать все основные особенности языка. Изучив главу 16, вы ознакомитесь с развитыми средствами ввода-вывода в языке C.

В главе 17, “Дополнительные возможности”, описаны те особенности языка, которые имеют более глубокий или скрытый смысл.

В главе 18, “Отладка программ”, показано, как можно использовать предпроцессор C для более удобной отладки разрабатываемых программ. В этой главе вы также познакомитесь с диалоговой отладкой. Популярный отладчик gdb был выбран для демонстрации этой техники отладки. Все прошлое десятилетие в мире программирования продолжалось обсуждение понятия «объектно-ориентированное программирование», или коротко ООП. Язык C не является объектно-ориентированным. Однако несколько других языков программирования, которые базируются на C, — это языки ООП.

В главе 19, “Объектно-ориентированное программирование”, дается краткое введение в ООП и его терминологию. Здесь также дан краткий обзор трех языков ООП, которые базируются на C, а именно: C++, C# и Objective-C.

В приложении А, “Кратко о языке программирования C”, приводятся краткие сведения о всех возможностях и структурах языка. Эту главу можно использовать как справочное пособие.

В приложении В, “Стандартная библиотека C”, описаны многие из стандартных библиотечных подпрограмм, которые вы найдете на всех системах, поддерживающих язык программирования C.

В приложении С, “Компиляция программ с помощью компилятора gcc”, описаны многие из обычно используемых вариантов при компиляции программ на языке C с помощью компилятора gcc, разработанного по лицензии GNU.

В приложении D, “Общие ошибки программирования”, вы найдете список наиболее часто встречающихся ошибок программирования.

Наконец, в приложении E, “Ресурсы”, приведен список ресурсов, к которым можно обратиться для получения дополнительной информации о языке программирования C и для его дальнейшего изучения.

Ответы на контрольные вопросы в конце глав можно найти на сайте: www.kochanwood.com. Эта книга не содержит никаких предположений о специфической компьютерной системе или операционной системе, на которой реализован язык программирования C. В книге представлено только краткое замечание о том, как компилировать и выполнять программы с помощью популярного компилятора для языка C — gcc. Я хочу поблагодарить следующих людей за советы при подготовке различных изданий моей книги: это Дуглас Маккормик, Джим Шарф, Генри Табикман, Дик Фритз, Стив Левай, Тони Ианайнно и Кен Браун. Я также хочу поблагодарить Генри Маллиша из Нью-Йоркского университета за его советы при написании книги и за его помощь в издании этой книги.

Более раннее издание этой книги было посвящено памяти Морин Коннеллай, бывшего технического редактора издательства Hayden Book Company, которая подготовила первый вариант этой книги.

Несколько основных принципов

В этой главе описаны отдельные фундаментальные понятия, знание которых необходимо при изучении программирования на языке C. Приведен краткий обзор методики программирования на языках высокого уровня и обсуждение процесса компилирования программы, разработанной на одном из таких языков.

Программирование

В действительности компьютеры не являются умными машинами, потому что они делают только то, что им приказывают. Большинство компьютерных систем выполняют операции на очень примитивном уровне. Например, большинство компьютеров умеют только суммировать числа или проверять значение на равенство нулю. Все основные операции любой компьютерной системы формируют базовый набор команд, который называют *системой команд* компьютера.

Чтобы решить задачу с помощью компьютера, вы должны представить решение задачи в терминах команд специфического компьютера. *Компьютерная программа* — это только набор команд, необходимых для решения определенной задачи. Подход или метод, который используется для решения задачи, известен как *алгоритм*. Например, если вы хотите разработать программу, которая проверяет, является ли число четным или нечетным, то вы пишете последовательность команд, которые решают задачу, после чего эта последовательность команд становится программой. Метод, который используется в целях проверки того, является ли число четным или нечетным, называется *алгоритмом*. Обычно, чтобы разработать программу для решения определенной задачи, решение сначала описывается в виде алгоритма, а затем на его основе разрабатывается программа, которая и выполняет этот алгоритм. Так, алгоритм для проверки числа на четность мог бы быть описан следующим образом.

Сначала число делится на два. Если остаток от деления является нулевым, число является четным, в ином случае число нечетное.

Имея логически правильный алгоритм, можно последовательно написать утверждения, необходимые для выполнения заданного алгоритма на специфической компьютерной системе. Эти утверждения должны быть выражены в командах определенного языка, такого как Visual Basic, Java, C++ или C.

Языки высокого уровня

Когда появились первые вычислительные устройства, то единственным способом их программирования было ручное введение в память устройства двоичных чисел, которые соответствовали определенным машинным командам. Следующее усовершенствование технологии ввода программы произошло с развитием *ассемблеров* — программ, которые давали возможность программисту работать с компьютером на несколько более высоком уровне. Вместо того, чтобы вводить последовательности двоичных чисел, составляющих программу для выполнения специфической задачи, ассемблер позволяет программисту использовать *мнемокоды*, или понятные аббревиатуры, которые преобразовываются в соответствующие машинные команды. С помощью мнемокодов можно также управлять расположением команд в памяти. Программа, называемая *ассемблером*, транслирует исходную программу на языке ассемблера из ее символического формата в соответствующие машинные команды компьютерной системы.

Поскольку в ассемблерах существует взаимно-однозначное соответствие между каждым утверждением ассемблера и соответствующей машинной командой, ассемблеры являются языками низкого уровня. Программист должен знать систему команд специфической компьютерной системы, чтобы написать программу на ассемблере, причем окончательная программа не является *переносимой*; т.е. программа не будет работать, если тип процессора отличается от того, для которого она была написана. Именно поэтому для различных типов процессоров разработаны собственные ассемблеры, которые учитывают специфику системы команд данного процессора. Программы, написанные на языке ассемблера, являются машинно-зависимыми.

Поэтому с развитием технологий программирования произошел закономерный переход к т.н. *языкам высокого уровня*, где одним из первых был язык FORTRAN (FORmula TRANslation). Программисты, разрабатывающие программы на языке FORTRAN, больше не должны были интересоваться архитектурой компьютера, а отдельные операции, выполняемые на FORTRAN, были намного более сложными по сравнению с командами ассемблера и не были связаны с системой команд специфической машины. Одна команда языка FORTRAN, или *утверждение*, приводили к выполнению многих отдельных машинных команд, в отличие от взаимно-однозначного соответствия между командами ассемблера и машинными командами.

Стандартизация синтаксиса языков высокого уровня подразумевала, что программа, написанная на языке высокого уровня, должна быть независима от типа компьютера. То есть программа могла работать на любой машине, которая поддерживала язык либо с небольшими изменениями, либо вообще без изменений. Чтобы поддерживать язык высокого уровня, должна быть разработана специальная компьютерная программа, которая переводит утверждения программы, написанной на языке высокого уровня, в форму, которую понимает компьютер, или другими словами, в специфические команды компьютера. Такая программа называется *компилятором*.

Операционные системы

Прежде чем продолжить разговор о компиляторах, необходимо понять роль компьютерной программы, известной как *операционная система*. Операционная система — эта программа, которая управляет всей компьютерной системой. Все операции ввода-вывода, которые выполняются на компьютере, проходят через операционную систему. Операционная система также управляет ресурсами компьютерной системы и обеспечивает выполнение пользовательских программ.

Одна из самых популярных операционных систем сегодня — это операционная система Unix, которая была разработана в лабораториях фирмы Bell. Unix — довольно уникальная операционная система, которая установлена на многих типах компьютерных систем, а ее различные модификации представляют и операционную систему Linux и Mac OS X. Обычно операционные системы традиционно связывались только с одним типом компьютерной системы. Но так как операционная система Unix была написана на языке программирования C и в ней не учитывалась архитектура компьютера, то она была довольно успешно перенесена на многие различные компьютерные системы.

Microsoft Windows XP — это пример другой популярной операционной системы, которая прежде всего рассчитана на работу с процессором Pentium (или совместимым с Pentium).

Компиляция программ

Компилятор является обычной программой, которая о основном никак не отличается от тех программ, которые представлены в этой книге, хотя и намного сложнее. Компилятор анализирует программу, написанную на определенном языке программирования, и затем переводит текст программы в код, необходимый для выполнения этой программы на вашей специфической компьютерной системе.

На рис. 2.1 показаны шаги, которые должны быть выполнены при вводе, компиляции и выполнении программы, разработанной на языке программирования C, и типичные команды Unix, которые должны быть введены в командной строке.

Программа, которая должна компилироваться, сначала записывается в виде файла на компьютерной системе. Имеются различные соглашения по именованию файлов, но в основном, выбор названия остается за вами. Исходной программе на языке C можно давать любое имя, но последними двумя символами должны быть ".c" (это не является обязательным требованием, это просто соглашение). Поэтому имя `prog1.c` является допустимым именем файла для программы на языке C в вашем компьютере.

Для написания программ на языке C обычно используется текстовый редактор. Например, популярным редактором является программа с именем `vi`, которая используется с операционными системами Unix. Программа на языке программирования C, которая записана в файл, называется *исходной программой*, или *исходным модулем*, поскольку представляет собой оригинальный текст программы. После того как исходная программа записана в файл, ее можно компилировать.

Процесс компиляции инициализируется с помощью ввода специальной команды. При вводе этой команды должно быть указано имя файла, который содержит исходную программу и который должен компилироваться. Например, в системе Unix имя команды, которая запускает процесс компиляции, состоит из двух символов `cc`. Если вы используете популярный компилятор GNU C, тогда команда на компиляцию будет иметь вид `gcc`.

Ввод строки

```
gcc prog1.c
```

запустит процесс компиляции исходной программы, которая содержалась в файле `prog1.c`. На первом шаге процесса компиляции компилятор анализирует каждое утверждение исходной программы и производит проверку на соответствие синтаксису и семантике языка.

Если на этом этапе компилятором обнаружены ошибки, то пользователь получает соответствующие сообщения сразу после процесса компиляции. В исходной программе ошибки должны быть исправлены и процесс компиляции должен быть повторен.

Типичные ошибки, которые происходят на этом этапе, — это незакрытые скобки (синтаксическая ошибка), или использование переменной, которая не определена (семантическая ошибка).

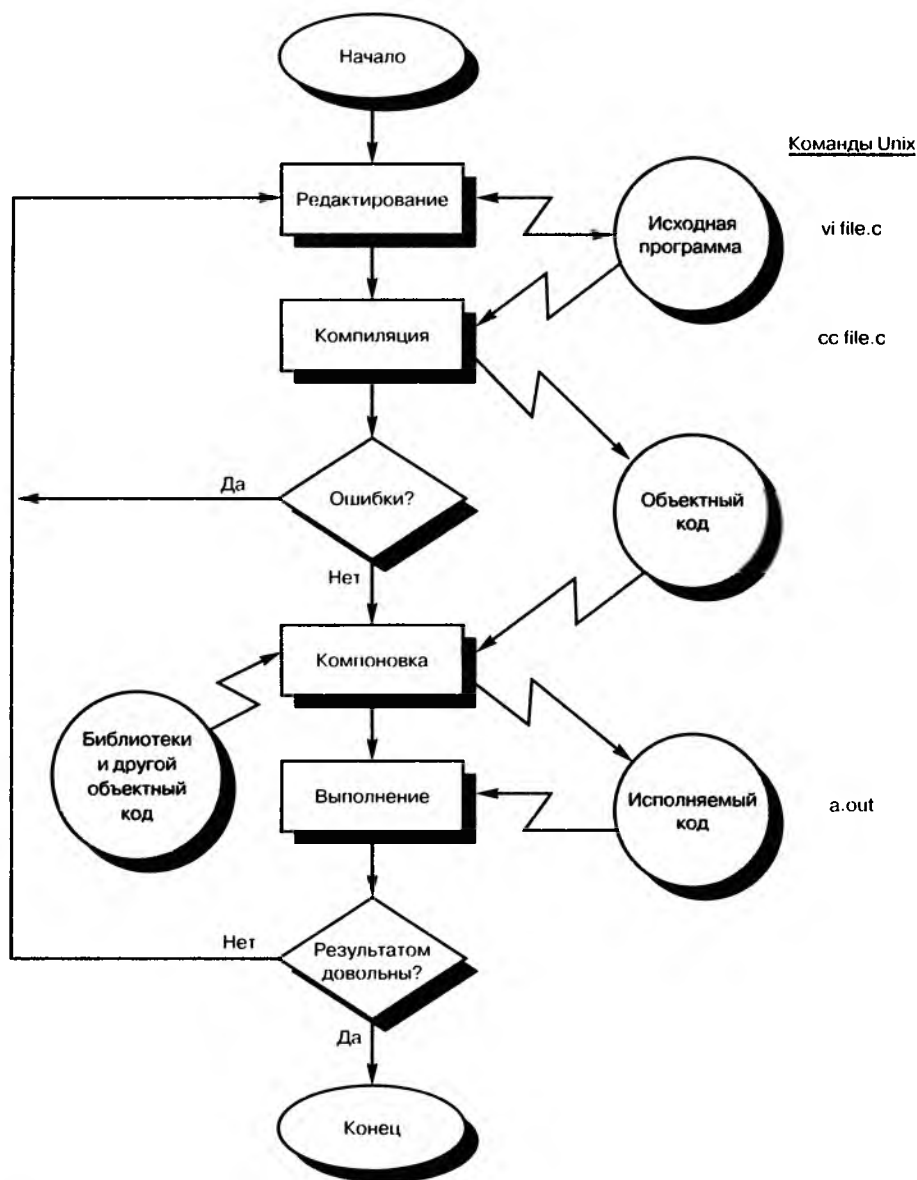


Рис. 2.1. Типичные шаги ввода, компиляции и выполнения программ на языке C при запуске из командной строки.

Когда все синтаксические и семантические ошибки в программе будут исправлены, компилятор начнет преобразовывать каждое утверждение программы в “более низкую” форму. На большинстве систем это означает, что каждое утверждение исходной

программы будет переведено компилятором в эквивалентные структуры на языке ассемблера, которые должны выполнять идентичные действия.

После того как программа была переведена в эквивалентную программу на языке ассемблера, следующий этап процесса компиляции переводит команды ассемблера в соответствующие машинные команды. На этом этапе может использоваться отдельный ассемблер, но обычно в большинстве систем программирования ассемблер встроен в компилятор.

Ассемблер анализирует каждую команду языка ассемблера и преобразовывает ее в двоичный формат. При этом формируется объектный код, который записывается в соответствующий файл. Этот файл обычно имеет то же самое имя, что и исходный файл, но с расширением ".o" (в системе Unix) вместо ".c". В операционной системе Windows символами расширения будут символы ".obj", которые заменяют символ ".c" в имени файла.

После того как исходная программа переводится в объектный код, ее необходимо скомпоновать. Этот процесс также выполняется автоматически каждый раз, когда команда `cc` или `gcc` набрана в командной строке системы Unix. Цель этапа компоновки состоит в том, чтобы получить законченную программу, готовую для выполнения на компьютере. Если исходная программа использует другие программы, которые были предварительно скомпилированы, то на этом этапе все программы объединяются. В объектные программы, которые используют библиотечные файлы системы, также будут сделаны вставки из библиотек во время выполнения этого этапа.

Процесс компиляции и объединения программ часто называют построением. Конечный объединенный файл, который является выполнимым форматом объектного кода, сохраняется в заключительном файле, который можно запустить или выполнить. В операционной системе Unix этот файл по умолчанию называют `a.out`. В операционной системе Windows исполняемый файл обычно имеет то же самое имя, что и исходный файл с расширением ".exe".

Для того чтобы впоследствии выполнить программу, все, что необходимо сделать, — это ввести имя исполняемого файла. Так, команда

```
a.out
```

произведет загрузку программы `a.out` в память компьютера и инициализирует ее выполнение.

Выполнение программы подразумевает, что последовательно выполняется каждое из утверждений программы. Если программа запрашивает некоторые данные от пользователя, что обычно называется вводом, то происходит временная приостановка выполнения программы таким образом, чтобы можно было ввести данные. Аналогично программа может реагировать на различные события, например нажатие мыши. Процесс отображения результатов работы программы называется выводом. Результаты появляются в окне, иногда называемом консолью. Вывод также можно сделать непосредственно в файл.

Если все выполняется правильно (вероятно, это не всегда будет получаться с первого раза), то можно считать, что разработка программы закончена. Если программа не выдает желаемые результаты, необходимо вернуться к исходной программе и повторно проанализировать логику алгоритма. Этот этап называется отладкой. Во время этого этапа определяются и устраняются все логические неточности и ошибки программы. При этом, вероятнее всего, необходимо будет сделать некоторые изменения в оригинальной программе, после чего полный процесс компиляции, компоновки и выполнения программы должны быть повторены, пока не будут получены правильные результаты.

Интегрированная среда разработки

Ранее были рассмотрены отдельные этапы создания исполняемой программы на языке С. При этом были приведены типичные команды, которые использовались на каждом этапе. Это этапы редактирования, компиляции, выполнения и отладки программ. Но обычно эти этапы объединены в единственном приложении, называемом *интегрированной средой разработки*, для обозначения которой часто используют аббревиатуру IDE (Integrated Development Environment). Интегрированная среда разработки – это программа на основе оконного интерфейса, которая позволяет легко управлять большими объектами, редактировать файлы, компилировать, компоновать, выполнять и отлаживать ваши программы.

Для Mac OS X, созданы две интегрированных среды разработки, CodeWarrior и Xcode, которые используются многими программистами. Что касается операционной системе Windows, то Microsoft Visual Studio – хороший пример удачной интегрированной среды разработки. Kylix – это популярная интегрированная среда разработки для создания приложений под Linux. Все интегрированные среды разработки значительно упрощают и сокращают полный процесс разработки и отладки программ, так что стоит потратить время на их освоение. Большинство интегрированных сред разработки также могут помочь в разработке программ на нескольких языках программирования в дополнение к С, таких как С# и С++. Для подробной информации об интегрированных средах разработки обратитесь к приложению Е, “Ресурсы”.

Интерпретаторы

Перед тем как закончить обсуждение процесса компиляции, заметим что есть и другой метод, используемый для анализа и выполнения программ, разработанных на языках высокого уровня. При использовании этого метода программы не компилируются, а интерпретируются. Интерпретатор анализирует и выполняет утверждения программы в одно и то же самое время. Этот метод обычно позволяет более легко отлаживать программы. С другой стороны, интерпретаторы обычно работают значительно медленнее, чем соответствующие компиляторы, потому что утверждения программы не преобразуются в машинные команды перед их выполнением.

БЕЙСИК и Java – это два языка программирования, в которых программы часто интерпретируются и не компилируются. Другие примеры включают оболочку системы Unix и язык Python. Можно также найти и интерпретаторы для языка программирования С.

Компиляция и запуск первой программы

В этой главе представлен общий обзор языка программирования С. Лучший способ понять конструкции языка С — это рассмотреть программы, написанные на этом языке. Для начала будет выбрана довольно простая программа, которая отображает в окне фразу “Программирование — это забавно”. В листинге 3.1 приведена программа на языке С, которая выполняет эту задачу.

Листинг 3.1. Создание первой программы на языке С

```
#include <stdio.h>
int main (void)
{
    printf ("Programming is fun.\n");
    return 0;
}
```

При программировании на языке С различаются строчные и заглавные буквы. Кроме того, в синтаксисе языка С не имеет значения, с какого места строки начинается ввод текста. Можно начать печатать ваши утверждения с любой позиции на строке. Это можно использовать для написания программ таким образом, чтобы их было легко читать. Отступы часто используются программистами как удобный способ для выделения фрагментов программы.

Компиляция

Для получения первой программы на языке С, сначала необходимо создать файл с этой программой. Для этой цели может использоваться любой текстовый редактор. Пользователи Unix часто используют редакторы, которые имеют имена `vi` или `emacs`.

Большинство компиляторов языка С распознают имена файлов, которые заканчиваются двумя символами “.с”, как программы, написанные на языке С. Предположим, что вы ввели программу из листинга 3.1 в файл с именем `prog1.c`. После этого программу необходимо откомпилировать.

При использовании компилятора GNU C, это можно сделать путем простого введения команды `gcc` в командной строке, сопровождаемой именем файла. Рассмотрим пример.

```
$ gcc prog1.c
$
```

Если Вы используете стандартный компилятор Unix C, то вместо символов `gcc` водится команда `cc`. В приведенном примере текст, который набирается в командной строке, выделен полужирным шрифтом. Знак доллара — это приглашение ко вводу команды, если вы будете компилировать вашу программу на языке C из командной строки. Приглашение ко вводу команды может быть представлено и некоторыми другими символами, кроме знака доллара.

Если вы сделали ошибку при вводе вашей программы, то компилятор перечислит их после того, как вы введете команду `gcc`, при этом будут указаны номера строк программы, которые содержат ошибки. Если вместо этой команды вновь появится приглашение ко вводу команды, как показано в предыдущем примере, то написанная программа будет отвечать всем синтаксическим правилам и не будет обнаружено никаких ошибок.

После компиляции и компоновки программы будет создана исполняемая версия программы. Если использовался компилятор GNU или стандартный компилятор языка C, то по умолчанию эта программа будет названа `a.out`. В операционной системе Windows исполняемый модуль часто получает имя `a.exe` вместо `a.out`.

Запуск программы

После получения исполняемого модуля программу можно запустить на выполнение, просто напечатав имя файла в командной строке.

```
$ a.out
Programming is fun.
$
```

Вы можете также определить любое другое имя для исполняемого файла во время компиляции программы. Для этого выбирается опция `-o` (символ "o"), которая сопровождается необходимым именем. Например, командная строка

```
$ gcc prog1.c -o prog1
```

приведет к компиляции программы `prog1.c` и создаст исполняемый файл с именем `prog1`, который может впоследствии быть выполнен, если набрать его имя.

```
$ prog1
Programming is fun.
$
```

Если при этом будет выведена ошибка, подобная следующей: "a.out: No such file or directory" (Нет такого файла или каталога), то, вероятнее всего, это означает, что в переменной окружения `PATH` нет текущего каталога. Вы можете либо добавить необходимый путь в переменную `PATH`, либо набрать команду в следующем виде: `./a.out`.

Рассмотрим первую программу

Рассмотрим вашу первую программу более подробно. Первая строка программы

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

означает, что заголовочный файл с именем `stdio.h` должен быть включен в начало программы, которую вы пишете. В нем содержится информация для компилятора о том, как выполнить вывод значений (процедура `printf`) из программы. Об этом мы поговорим позже в главе 13. “Предпроцессор”, где эти вопросы обсуждаются более подробно. Строка программы, которая выглядит как

```
int main (void)
```

сообщает системе, что именем программы является `main` и что эта программа *возвращает* целое число, о чем говорит аббревиатура “`int`”. Имя `main` — это специальное имя, которое указывает, где программа должна начать выполнение. Круглые скобки сразу после слова `main` свидетельствуют о том, что это имя функции. Ключевое слово `void`, которое находится в круглых скобках, означает, что в функцию `main` не передается никаких аргументов. Эти понятия детально объясняются в главе 8, “Работа с функциями”. Теперь, когда вы определили функцию, вы должны написать, что именно эта функция должна выполнять. Это можно сделать, разместив все утверждения программы в паре фигурных скобок. Все утверждения программы, заключенные в фигурные скобки, будут относиться к функции `main`. В листинге 3.1 записано только два таких утверждения. Первое утверждение определяет, что должна быть вызвана подпрограмма с именем `printf`. В качестве аргумента в подпрограмму `printf` будет передана последовательность символов.

```
"Programming is fun.\n"
```

Функция `printf` находится в библиотеке компилятора C, и она просто печатает или отображает те аргументы, которые были подставлены вместо параметров, как вскоре вы это и увидите на экране вашего компьютера. Последние два символа в строке, а именно обратная наклонная черта “\” и символ “n”, вместе составляют символ `newline` (новая строка). При этом система должна сделать то, о чем говорит название символа — выполнить переход на новую строку. Любые символы, которые будут напечатаны после символа `newline`, появятся на следующей строке дисплея. Фактически символ `newline` подобен клавише возврата каретки на пишущей машинке. Запомните это!

Все утверждения в программе на языке C должны заканчиваться точкой с запятой (;). Именно поэтому точки с запятой должны быть сразу поставлены после закрывающей круглой скобки при вызове функции `printf`.

Последнее утверждение, написанное как

```
return 0;
```

говорит о том, что выполнение функции закончено и что в систему возвращается значение 0. При этом вместо нуля можно использовать любое целое число. Ноль используется в соответствии с соглашением об индикации успешного завершения программы, т.е. это говорит о том, что при выполнении программы не произошло никаких ошибок. При этом могут использоваться различные числа для указания различных типов ошибок, которые произошли (например, не найден необходимый файл).

Проверка значения возврата может производиться другими программами (например, оболочкой Unix), чтобы определить, выполнена ли программа успешно.

Теперь, когда вы закончили анализировать вашу первую программу, вы можете изменить ее так, чтобы отобразить фразу: "And programming in C is even more fun". (А программирование на языке C является еще более забавным). Это можно сделать простым добавлением еще одной функции `printf` к подпрограмме, как показано в листинге 3.2.

Помните, что каждое утверждение программы C должно заканчиваться точкой с запятой.

Листинг 3.2

```
#include <stdio.h>
int main (void)
{
    printf ("Programming is fun.\n");
    printf ("And programming in C is even more fun.\n");

    return 0;
}
```

Если вы введете программу из листинга 3.2 в файл, а затем откомпилируете и выполните ее, то увидите следующие строки в окне монитора, иногда называемом консолью.

Листинг 3.2. Вывод

```
Programming is fun.
And programming in C is even more fun.
```

Как вы можете увидеть в следующем примере, в программе не нужно делать отдельный вызов функции `printf` для вывода каждой строки. Рассмотрите программу, показанную в листинге 3.3, и попробуйте предсказать результаты вывода перед запуском программы.

Листинг 3.3. Отображение нескольких строк

```
#include <stdio.h>
int main (void)
{
    printf ("Testing...\n..1\n...2\n....3\n");

    return 0;
}
```

Листинг 3.3. Вывод

```
Testing...
..1
...2
....3
```

Отображение значений переменных

Функция `printf` очень часто будет использоваться в этой книге, поскольку она обеспечивает простой и удобный способ для отображения результатов работы программы. Мало того, что могут быть отображены простые фразы, как мы уже делали это раньше, но также можно отображать и значения *переменных* и результаты вычислений. Программа, показанная в листинге 3.4, использует функцию `printf` для отображения результатов сложения двух чисел, а именно чисел 50 и 25.

Листинг 3.4. Отображение переменных

```
#include <stdio.h>
int main (void)
{
    int sum;

    sum = 50 + 25;
    printf ("The sum of 50 and 25 is %i\n", sum);

    return 0;
}
```

Листинг 3.4. Выход

```
The sum of 50 and 25 is 75
```

В листинге 3.4 в первом утверждении *объявляется* переменная `sum`, которая имеет *тип* `int`. В языке C требуется, чтобы все переменные были объявлены прежде, чем они будут использованы в программе. Объявление переменной сообщает компилятору языка C, как именно будет использоваться отдельная переменная в данной программе. Эта информация необходима компилятору для генерирования команд, с помощью которых будут храниться и извлекаться значения для данной переменной. Переменная, объявленная с ключевым словом `int`, может использоваться только для хранения целочисленных значений, т.е. чисел без дробной части. Примеры целочисленных значений: - 3, 5, -20 и 0. Числа с дробной частью, такие как 3.14, 2.455 и 27.0, называются *вещественными числами* или *числами с плавающей запятой*.

Целочисленная переменная `sum` используется для хранения результатов сложения двух целых чисел 50 и 25. Пустая строка после объявления этой переменной оставлена преднамеренно, для визуального отделения объявления переменных программы от утверждений программы. Это не обязательно, но лучше оформлять программу в соответствии с требованиями стиля. Иногда добавление отдельной пустой строки в программе может сделать программу более читаемой.

Утверждение программы

```
sum = 50 + 25;
```

выглядит как и в большинстве других языков программирования: число 50 суммируется (о чем говорит знак плюс "+") с числом 25, и результат сохраняется (что делается с помощью *оператора присваивания*, который выражен знаком "=") в переменной `sum`.

Вызов функции `printf` в листинге 3.4 записан с двумя *параметрами*, заключенными в круглые скобки. Эти параметры разделяются запятой. Первый параметр – это символьная строка, которая будет отображена на дисплее. Однако наряду с символьной

строкой вы также должны отобразить значения некоторых переменных программы. В нашем случае будет отображено значение переменной `sum`, которое будет выведено на экране символов

Будет отображена сумма чисел 50 и 25. Для этого в строке, которая используется в качестве первого параметра, поставлен символ процента — специальный символ, который распознается функцией `printf`. Символ, который непосредственно следует за знаком процента, определяет, какое значение должно быть отображено в этом месте. В предыдущей программе, символ `i` будет распознан функцией `printf` как команда отображать значение как целое число.

Всякий раз, когда функция `printf` находит символы `"%i"` в символьной строке, она отображает значение очередного параметра функции. Поскольку именно `sum` и будет очередным параметром функции `printf`, его значение и будет отображено после того, как будет отображена строка `"The sum of 50 and 25 is"`.

Теперь пробуйте предсказать вывод, который будет сделан программой, показанной в листинге 3.5.

Листинг 3.5. Отображение нескольких значений

```
#include <stdio.h>
int main (void)
{
    int value1, value2, sum;

    value1 = 50;
    value2 = 25;
    sum = value1 + value2;
    printf ("The sum of %i and %i is %i\n", value1, value2, sum);

    return 0;
}
```

Листинг 3.5. Вывод

```
The sum of 50 and 25 is 75
```

Обратите внимание, что функция `printf` также позволяет использовать для отображения целого числа символы управления форматом `"%d"`. Во всех последующих главах этой книги для форматирования вывода будет использоваться символьная последовательность `"%i"`.

В первом утверждении программы объявлены три переменные, названные `value1`, `value2` и `sum`. Все они имеют тип `int`. Это утверждение будет эквивалентно трем строкам, где будут написаны три утверждения следующим образом.

```
int value1;
int value2;
int sum;
```

После того как эти три переменные были объявлены, программа присваивает значение 50 для переменной `value1` и затем присваивает значение 25 переменной `value2`. После этого вычисляется сумма этих двух переменных и результат присваивается переменной `sum`. При обращении к функции `printf` теперь записано четыре аргумента. Повторю еще раз, что первый аргумент, обычно называемый *строкой*

форматирования, указывает системе, как остающиеся аргументы должны быть отображены. Значение переменной `value1` должно быть отображено непосредственно после вывода строки "The sum of." Точно так же значения для переменных `value2` и `sum` должны быть отображены в соответствующих местах, где в строке форматирования встречаются символы "%i".

Комментарии

В заключительной программе этой главы (листинг 3.6) вводится понятие комментария. Комментарии используются в программе для документации программы и делают ее более понятной. Как вы увидите в следующем примере, комментарии служат для объяснения пользователю программы — программисту или тому, кто желает ознакомиться с программой, или тому, кто будет обслуживать программу, — что именно программист имел в виду, когда он писал специфическую программу или специфическую последовательность утверждений.

Листинг 3.6. Использование комментариев в программе

```
/* Эта программа складывает два целых числа
и отображает результат на экране */
#include <stdio.h>
int main (void)
{
    // Описываем переменные
    int value1, value2, sum;
    // Определяем значения и рассчитываем сумму
    value1 = 50;
    value2 = 25;
    sum = value1 + value2;
    // Отображаем результат
    printf ("The sum of %i and %i is %i\n", value1, value2, sum);
    return 0;
}
```

Листинг 3.6. Вывод

```
The sum of 50 and 25 is 75
```

Есть два способа вставить комментарии в программу на языке C. Каждому отдельному комментарию должны предшествовать два символа `/*` и `*/`. Чтобы закончить комментарий, используются символы `*/` и `/*`, между которыми нельзя вставлять пробел. Все символы, заключенные между началом (`/*`) и концом (`*/`) комментария, игнорируются компилятором языка C.

Эта форма комментария часто используется в том случае, когда для комментария необходимо несколько строк программы. Второй способ добавлять комментарий к программе — использовать два последовательных символа наклонной черты вправо (`//`). Все, что следует за этими наклонными чертами вплоть до конца строки, игнорируется компилятором.

В программе (листинг 3.6) использовались четыре отдельных утверждения для комментариев. Эта программа идентична ранее приведенной программе и является несколько надуманным примером, потому что только первый комментарий в начале программы объясняет суть и действительно необходим. (В программу можно вставить

сколь угодно много комментариев, но в нашем случае удобочитаемость программы фактически ухудшилась вместо того, чтобы стать лучше!)

Грамотное использование комментариев в программе подразумевает, что они должны только разъяснять плохо понимаемые фрагменты программы, причем их не должно быть слишком много. Программист будет много раз возвращаться к программе, которую он составил, возможно, только шесть месяцев назад, и вдруг он с тревогой обнаруживает, что он не помнит задачи отдельной подпрограммы или специфической группы утверждений. Простой комментарий, грамотно вставленный в программу, возможно, сэкономит много времени, которое может быть потрачено впустую при восстановлении прежней логики подпрограммы или группы утверждений.

Это должно войти в привычку — вставлять комментарии в программу во время ее написания, поскольку именно в этот период намного проще ее документировать.

Кроме того, комментарии пригодятся и на этапе отладки, когда определяются и устраняются ошибки в логике программы. Комментарии помогут не только в уточнении логических связей программы, но и помогут указать источник логической ошибки. Наконец, я должен сказать, что не всем программистам нравится документировать программу во время разработки. Но фактически после того, как вы закончите отлаживать вашу программу, вы вряд ли захотите возвращаться к программе, чтобы вставить комментарии. Вставка комментариев при разработке будет много проще и эффективнее, чем дополнительный этап работы по оформлению программы.

На этом заканчивается вводная часть курса по разработке программ на языке C. К этому времени вы уже должны хорошо представлять, какие средства используются при написании программ на C, и уметь разрабатывать небольшие программы самостоятельно. В следующей главе вы изучите более сложные понятия этого мощного и гибкого языка программирования. Но сначала проверьте ваши знания, выполнив следующие упражнения.

Упражнения

Наберите и выполните шесть программ, представленных в этой главе. Сравните вывод, сделанный каждой программой, с выводом, представленным после каждой программы в книге.

1. Напишите программу, которая выводит следующий текст.
 1. В языке C все символы нижнего регистра значимы.
 2. Слово `main` указывает на начало выполнения программы.
 3. Открывающая и закрывающая фигурные скобки содержат утверждения программы.
 4. Все утверждения программы должны заканчиваться точкой с запятой.
2. Что будет выведено в результате выполнения следующей программы?

```
#include <stdio.h>
int main (void)
{
    printf ("Testing...");
    printf ("....1");
    printf ("...2");
```

```
printf ("..3");
printf ("\n");
return 0;
}
```

3. Напишите программу, которая вычитает число 15 из 87 и отображает результат с соответствующим сообщением на экране.
4. Найдите синтаксические ошибки в следующей программе. Введите и выполните исправленную программу и убедитесь, что вы нашли все ошибки.

```
#include <stdio.h>
int main (Void)
{
    INT sum;
    /* Подсчитать результат
    sum = 25 + 37 - 19
    /* Отобразить результат //
    printf ("The answer is %i\n" sum);
    return 0;
}
```

5. Какой вывод будет сделан в результате выполнения следующей программы?

```
#include <stdio.h>
int main (void)
{
    int answer, result;
    answer = 100;
    result = answer - 10;
    printf ("The result is %i\n", result + 5);
    return 0;
}
```


Переменные, типы данных и арифметические выражения

В этой главе вы узнаете больше о переменных, именах и константах. Также будут детально рассмотрены основные типы данных и некоторые фундаментальные правила, которые используются при написании арифметических выражений на языке C.

Работа с переменными

Первые программисты должны были писать программы в двоичных кодах, используя систему команд того компьютера, для которого они программировали. При этом подразумевалось, что команды должны были быть закодированы программистом вручную в виде двоичных чисел, прежде чем они будут введены в компьютер. Кроме того, программист должен был явно обращаться к памяти, указывая непосредственные адреса памяти или ссылки для всех переменных, сохраняемых в памяти компьютера.

Современные языки программирования позволяют в большей степени концентрироваться на логической стороне проблемы, возникающей при решении определенных задач, и в меньшей степени думать о машинных кодах и памяти. Современные языки программирования позволяют присваивать переменным понятные символические имена и использовать их при вычислениях и хранении данных. С именами переменных связывается тип данных, который контролируется компилятором и для которого выделяется определенное количество байтов памяти.

В примерах к главе 3, “Компиляция и запуск первой программы”, были использованы несколько переменных для хранения целочисленных значений. Например, в листинге 3.4 была использована переменная `sum` для хранения суммы двух целых чисел 50 и 25. В языке программирования C разрешены и другие типы данных, которые необходимо присвоить переменным до того, как они будут использованы в программе. Переменные могут обозначать числа с плавающей запятой, символы и даже указатели на определенные ячейки памяти.

Правила для написания имен переменных очень простые. Имя должно начинаться с буквы или символа подчеркивания (`_`), за которыми могут следовать любые комбинации букв в любом регистре, символы подчеркивания или цифры 0–9. Ниже показан перечень разрешенных имен.

```
sum
pieceFlag
i
J5x7
Number_of_moves
_sysflag
```

Но следующие имена уже не будут являться правильными:

```
sum$value$ — включен недопустимый символ;
piece flag — пробел не разрешен;
3Spencer — имя переменной не должно начинаться с числа;
int — int является зарезервированным словом.
```

Слово "int" не может использоваться как имя переменной, поскольку оно имеет специальное значение и компилятор связывает с ним определенные действия. Такие слова называются *зарезервированными* или *ключевыми* словами. Они распознаются компилятором C как директивы и поэтому не могут использоваться в качестве имен переменных.

В приложении А, "Справочник по языку программирования C", приведен полный список зарезервированных слов. Необходимо всегда помнить, что в языке программирования C различаются заглавные и строчные буквы. Поэтому имена переменных sum, Sum и SUM будут ссылаться на различные переменные. Длина имени может быть сколь угодно большой, но только первые 63 символа будут учитываться, или будут являться значимыми, а в некоторых специальных случаях, описанных в приложении А, только первый набор из 31 символа будет значимым. Но обычно на практике не используются такие длинные имена, поскольку их довольно неудобно набирать и при этом ухудшается зрительное восприятие программы. Например, хотя следующая строка и будет являться правильным утверждением

```
theAmountOfMoneyWeMadeThisYear = theAmountOfMoneyLeftAttheEnd -
    OfTheYeartheAmountOfMoneyAtTheStartOfTheYear;
```

она плохо воспринимается и затрудняет чтение программы. Гораздо лучше написать следующим образом.

```
moneyMadeThisYear = moneyAtEnd - moneyAtStart;
```

При этом будет занято меньше места и более четко выделено назначение отдельных переменных.

Когда выбираете имя переменной, не спешите и придерживайтесь следующих правил. Выбирайте имя, которое четко определяет назначение переменных. Причина здесь очевидна. Вместе с комментариями, хорошо продуманные имена переменных значительно улучшают наглядность программы, что способствует лучшему пониманию логики программ и поможет сэкономить много времени при дальнейшей работе над программой или при ее отладке. При этом можно будет значительно сократить описательную часть программы, т.к. программа будет хорошо восприниматься. О хорошо понимаемых программах говорят, что они являются *самодокументированными*.

Типы данных и константы

Вы уже познакомились с базовым типом данных `int`, и, как вы помните, переменные, объявленные как тип `int`, могут использоваться только для хранения целочисленных значений. Это означает, что в таких переменных не выделено места для хранения дробной части.

В языке программирования C используются еще четыре базовых типа данных: `float`, `double`, `char` и `_Bool`. Переменные, объявленные как тип `float`, могут использоваться для хранения вещественных чисел (чисел с плавающей запятой). Переменные типа `double` также используются для хранения вещественных чисел, но удвоенной точности. Переменные типа `char` могут использоваться для хранения отдельных символов, таких как буквы, цифры или точки с запятой. Наконец, тип `_Bool` используется только для хранения двух значений: 0 или 1.

Переменные этого типа используются в тех ситуациях, когда требуется хранить только один из двух вариантов: да/нет, включено/выключено, правда/ложь. В языке программирования C любое число, отдельный символ или строка символов считаются *константами*. Например, число 58 представляет целочисленное значение, которое является константой. Строка символов "Программирование на языке C забавно.\n" также является константой.

Выражение, состоящее исключительно из констант, является *константным выражением*. Поэтому выражение

```
128 + 7 - 17
```

является константным выражением, поскольку каждый элемент этого выражения является константой. Но если будет присутствовать переменная `i`, объявленная как тип `int`, то выражение

```
128 + 7 - i
```

уже не будет являться константным.

Базовый тип `int`

В языке программирования C целочисленная константа состоит из одной цифры или последовательности нескольких цифр. Предшествующий цифрам знак минус говорит о том, что значение является отрицательным. Значения 158, -10 и 0 являются примерами целочисленных констант. Никаких пробелов между цифрами не допускается и значения больше 999 не могут форматироваться с помощью запятой (Поэтому значение 12,000 не является правильной целочисленной константой и должно быть записано как 12000).

В языке C для отображения целочисленных констант можно использовать не только базовое число 10 (десятичные числа). Если первое число является 0, то это значит, что данное число рассматривается как восьмеричное число (базовое число 8). В этом случае все остальные числа должны составлять допустимые восьмеричные числа, т.е. быть числами из диапазона чисел 0-7. Таким образом, для того, чтобы задать восьмеричное число 50, которое эквивалентно десятичному числу 40, необходимо написать 050.

Аналогичным образом, восьмеричное число 0177 равно десятичному числу 127 ($1 \cdot 64 + 7 \cdot 8 + 7$). Целочисленное значение может отображаться в восьмеричной нотации с помощью *символа форматирования* "%o", который ставится в строке форматирования в утверждении вывода. В этом случае отображается восьмеричное значение без

предшествующего нуля. Символ форматирования “%#o” приводит к отображению восьмеричного значения с предшествующим нулем.

Если целочисленной константе предшествует нуль и символ X (независимо в каком регистре), то целочисленная константа считается шестнадцатеричным значением (базовое число 16). Символы, непосредственно следующие за символом X, и составляют само число. Эти символы могут быть числами от 0 до 9, а также буквами от A до F (a-f). Буквы представляют значения от 10 до 15 соответственно. Поэтому для присваивания целочисленного шестнадцатеричного значения FFEF0D переменной rgbColor, необходимо написать следующее утверждение.

```
rgbColor = 0xFFEF0D;
```

Символ форматирования “%x” используется для отображения значений в шестнадцатеричном виде без предшествующих символов “0x” и букв a-f в нижнем регистре. Для отображения значения с предшествующими символами “0x”, необходимо использовать символ форматирования “%#x”, как показано в следующем примере.

```
printf ("Color is %#x\n", rgbColor);
```

Для отображения символов в верхнем регистре, необходимо использовать символы форматирования “%X” и “%#X”.

Хранение и диапазоны

Любое значение, будь то символы, целые или вещественные числа, имеет связанный в него диапазон значений. Этот диапазон зависит от количества памяти, выделяемой для хранения каждого типа данных, и не определяется в языке программирования C.

Обычно этот диапазон зависит от компьютера, на котором вы работаете, и поэтому реализация кода называется машинно-зависимой. Например, целочисленное значение может занимать 32 бита на вашем компьютере и 64 бита на другом компьютере. Вы никогда не должны писать программу, в которой учитываются размеры типов данных, хотя при этом гарантируется, что минимальное значение занимаемой памяти будет одинаковым для всех типов данных на всех компьютерах. Например, гарантируется, что целочисленное значение типа `int` сохраняется как минимум в 32 битах, что соответствует размеру слова (word) на большинстве компьютеров. (см. табл. A.4 в приложении A для получения более полной информации о типах данных).

Вещественный тип float

Переменные, объявленные как тип `float`, могут использоваться для хранения вещественных чисел, т.е. чисел с плавающей запятой. Именно для этих целей и введен тип `float`. Вы можете не ставить цифры до или после точки, но, разумеется, то в каком-то месте цифры должны стоять. Значения 3., 125.8 и -0.001 являются правильными примерами задания вещественных чисел. Для отображения вещественных чисел в процедуре `printf` используется символ форматирования “%f”.

Вещественные константы могут также отображаться в научной нотации. Константа 1.7e4 является вещественным числом, выраженным в научной нотации и представляющим значение $1.7 \cdot 10^4$.

Значение перед буквой *e* называется *мантиссой*, тогда как значение после буквы *e* называется *показателем степени*. Показатель степени, который может быть со знаком плюс или минус, представляет число, в которое должно быть возведено число 10 и на которое должна быть умножена мантисса. Поэтому константа 2.25e-3 представляет

собой значение мантиссы 2.25, умноженное на 10^{-3} (число -3 является показателем степени), т.е. число 0.00225.

Символ `e`, который разделяет мантиссу и показатель степени, может быть написан как в верхнем, так и в нижнем регистре.

Для отображения на экране вещественных чисел в научной нотации используется символ форматирования `%e`. При этом значения могут отображаться как в виде чисел с плавающей запятой, так и в виде чисел в научной нотации. Если показатель степени меньше -4 или больше 5, то удобнее использовать научную нотацию, в противном случае надо использовать символ форматирования `%f`. При использовании этого символа форматирования получается наиболее наглядное отображение числа.

Для получения шестнадцатеричных чисел используются префиксы `0x` или `0X`, непосредственно за которыми располагается мантисса числа, за мантиссой идет символ `r` или `R` с последующим числом, которое представляет степень числа 2 и может иметь знак плюс или минус. Например, запись вида `0x0.3p10` представляет десятичное число $192 (3/16 * 2^{10})$.

Числа удвоенной точности `double`

Числа удвоенной точности подобны числам типа `float`, но используются в тех случаях, когда недостаточно точности чисел типа `float`. Переменные, объявленные как тип `double`, могут сохранять вдвое больше цифр, чем переменные типа `float`. В большинстве компьютеров переменные типа `double` занимают 64 бита.

Если специально не оговорено, то все вещественные константы трактуются компилятором C как переменные типа `double`. Для того чтобы вещественную переменную объявить как тип `float`, необходимо добавить символ `f` или `F`.

12.5f

Для отображения переменной типа `double` на экране, используются символы форматирования `%f`, `%e` или `%g`, которые используются и при отображении вещественных чисел.

Символьный тип `char`

Переменные типа `char` можно использовать для хранения отдельных символов. Символьные константы объявляются с помощью отдельных символов, заключенных в одиночные кавычки. Поэтому записи вида `'a'`, `','` и `'0'` являются примерами правильно заданных символьных констант. В первом случае задается буква `a`, во втором случае это точка с запятой и в третьем случае это число 0. Не путайте символьные константы, которые представляют один символ, со *строкой символов*, которая представляет один или несколько символов, заключенных в двойные кавычки.

Символьная константа `'\n'` (символ новой строки) является допустимым символом, хотя по внешнему виду не соответствует ранее приведенному правилу. Но это правильная запись, т.к. символ обратной черты `'\'` для компилятора языка C не является действительно учитываемым символом, а имеет специальное назначение.

Другими словами, компилятор языка C трактует запись `'\n'` как один символ, хотя в действительности записано два символа. Но при этом наличие обратной черты заставляет компилятор языка C по-другому воспринимать символ `n`. Более подробно об этом рассказывается в приложении А.

В приложении А обсуждаются методы хранения символов из расширенного набора символов, использование управляющих символьных последовательностей, универсальных и расширенных символов.

Символ форматирования “%с” может использоваться для отображения значений символьных переменных на экране.

Булев тип `_Bool`

Переменные типа `_Bool` предназначены для хранения только двух значений, 0 и 1. Размер памяти, выделяемый для хранения каждой переменной, не оговаривается. Переменные типа `_Bool` используются для хранения результатов булевых вычислений. Например, переменные такого типа можно использовать для указания состояния режима чтения их файла, т.е. закончено или нет чтение из файла.

По соглашению, 0 используется для указания состояния “ложь”, а 1 используется для состояния “истина”. При присваивании значения переменной типа `_Bool`, 0 записывается как 0, а любое значение, отличное от 0, записывается как единица.

Для более удобной работы с переменными типа `_Bool`, в стандартном заголовочном файле `stdbool.h` для этих переменных определены значения `true` и `false`. Пример их использования показан в листинге 6.10. (в главе 6 “Вопросы точности”).

В листинге 4.1 показано использование основных типов данных языка C.

Листинг 4.1. Использование основных типов данных.

```
#include <stdio.h>
int main (void)
{
    int integerVar = 100;
    float floatingVar = 331.79;
    double doubleVar = 8.44e+11;
    char charVar = 'W';
    _Bool boolVar = 0;

    printf ("integerVar = %i\n", integerVar);
    printf ("floatingVar = %f\n", floatingVar);
    printf ("doubleVar = %e\n", doubleVar);
    printf ("doubleVar = %g\n", doubleVar);
    printf ("charVar = %c\n", charVar);
    printf ("boolVar = %i\n", boolVar);

    return 0;
}
```

Листинг 4.1 Вывод.

```
integerVar = 100
floatingVar = 331.790009
doubleVar = 8.440000e+11
doubleVar = 8.44e+11
charVar = W
boolVar = 0;
```

В первом утверждении из листинга 4.1 объявлена целочисленная переменная `integerVar` (о чем говорит ключевое слово `int`), и ей присвоено значение 100. Это же самое можно сделать и с помощью двух следующих утверждений.

```
int integerVar;  
integerVar = 100;
```

Обратите внимание, что во второй строке, выведенной программой на экран, значение 331.79, которое было присвоено переменной `floatingVar`, отображается как 331.790009. На самом деле, действительное значение, которое отображается, зависит от особенностей системы, которая используется для вычислений. Причина такой “неаккуратности” кроется в особенностях хранения чисел в компьютере. Наверное, вы неоднократно обращали внимание на результаты вычислений, производимых карманным калькулятором, когда, например, вы делите 1 на 3 и получаете результат .33333333. Это происходит в результате округления результата, так как теоретически должно быть получено бесконечное число троек. Но калькулятор может сохранять только определенное число цифр, которые и определяют точность вычислений. То же самое происходит и в компьютере. Некоторые вещественные числа не могут быть точно сохранены в памяти.

Для отображения переменных типа `float` или `double` можно выбрать один из трех форматов. Символ форматирования “%f” используется для вывода значения в наиболее привычном виде.

Если специально не оговорено, то вещественные числа отображаются округленными с шестью цифрами после запятой. Позже в этой главе вы узнаете, как делать вывод с различным числом цифр.

Символ форматирования “%e” используется для отображения значений вещественных чисел в научной нотации. Как и ранее, только шесть цифр по умолчанию отображаются на экране.

Если используется символ форматирования “%g”, то производится выбор между форматами “%f” и “%e” и при этом удаляются все последующие нули. Если нет цифр после запятой, то дробная часть не отображается совсем.

В предпоследнем утверждении вывода, символ форматирования “%c” используется для отображения одного символа ‘W’, который был присвоен переменной `charVar` при объявлении переменной. Запомните, что поскольку любая символьная строка (такая как первый аргумент процедуры `printf`) выделяется двойными кавычками, то и символьная константа должна заключаться в двойные кавычки.

В последней процедуре `printf` показан вывод переменной типа `_Bool`, значение которой отображается с помощью символа форматирования для целых чисел “%i”.

Спецификаторы типов: `long`, `long long`, `short`, `unsigned`, `signed`

Если при объявлении переменной спецификатор типа помещен непосредственно перед ключевым словом `int`, то на некоторых компьютерах объявленная переменная будет иметь расширенный диапазон значений. Объявление для расширенного диапазона чисел можно сделать следующим образом.

```
long int factorial;
```

Такое объявление переменной `factorial` задает целочисленную переменную с расширенным диапазоном. Но как и при объявлении переменных типа `floats` и `doubles`, действительный диапазон переменных зависит от используемого типа компьютера.

На многих компьютерах типы `int` и `long int` имеют один и тот же диапазон и для их хранения выделяется 32 бита ($2^{31}-1$ или 2,147,483,647).

При объявлении типа `long int` к задаваемому значению необходимо добавить букву `L` (в верхнем или нижнем регистре). При этом недопустимы пробелы между цифрами и буквой. Поэтому следующее объявление

```
long int numberOfPoints = 131071100L;
```

создаст переменную `numberOfPoints` и присвоит ей начальное значение 131,071,100.

Для отображения переменной типа `long int` с помощью процедуры `printf`, в качестве модификатора используется буква `l`, которая размещается перед символами форматирования `i`, `o` и `x`. Таким образом, символ форматирования `%li` необходимо использовать для отображения переменных типа `long int` в десятичном формате. Символ форматирования `%lo` используется для отображения переменных с расширенным диапазоном значений в восьмеричном формате, и символ форматирования `%lx` — для отображения в шестнадцатеричном формате.

Также можно объявить тип `long long int`, при этом объявление переменной как

```
long long int maxAllowedStorage
```

создаст переменную для хранения расширенного диапазона значений, при котором будет использоваться по крайней мере 64 бита. При выводе значений типа `long long int` в символе форматирования необходимо использовать две буквы `l`, например `%lli`.

Спецификатор `long` можно использовать и при объявлении переменных типа `double`, как показано ниже.

```
long double US_deficit_2004;
```

Константа типа `long double` записывается как константа типа `floating` с добавленной буквой `l` или `L` сразу после цифр.

```
1.234e+7L
```

Для отображения переменных типа `long double`, используется модификатор `L`. Поэтому символ форматирования для вещественных переменных с расширенным диапазоном значений будет иметь вид `%Lf`. Если использовать символ форматирования `%Le`, то эта же переменная будет отображена в научной нотации, а при использовании символа форматирования `%Lg` автоматически будет сделан выбор между выводом в формате `%Lf` или в формате `%Le`.

Спецификатор `short`, будучи помещен перед ключевым словом `int`, указывает компилятору `C` на то, что данная переменная объявлена для хранения небольших целочисленных значений. Причиной использования таких переменных может быть желание сэкономить память, когда в некоторых ситуациях используется значительное число переменных с небольшими значениями, но при этом ограничен ресурс памяти.

В некоторых компьютерах размер типа `short int` будет занимать в два раза меньше битов, чем для типа `int`. Но в любом случае гарантируется, что количество выделяемых битов для переменных типа `short int` будет не меньше 16 бит.

Для задания переменных типа `short int` в языке `C` нет специального формата. Для отображения переменных типа `short int` используется модификатор `h`, который помещается перед символом форматирования: `%hi`, `%ho` или `%hx`. Но можно использовать и обычное форматирование для целочисленных переменных при отображении типа `short int`, при этом формат `short int` будет преобразован в формат `int`.

Спецификатор `unsigned` помещается перед ключевым словом `int` при объявлении переменных целочисленного типа и указывает компилятору на то, что задается положительное число. Это число обычно используется как счетчик и не может иметь отрицательное значение.

Используя целочисленные переменные типа `unsigned` для хранения положительных чисел, можно расширить диапазон используемых значений. Константа типа `unsigned int` задается с помощью буквы `u` (или `U`), которая помещается после цифр, как показано ниже.

```
0x00ffU
```

Можно комбинировать буквы `u` (или `U`) и `l` (или `L`) при задании целочисленных констант, при этом запись

```
20000UL
```

указывает компилятору на то, что константа `20000` имеет тип `unsigned long`.

Целочисленная константа, за которой не следуют буквы `u`, `U`, `l` или `L`, но которая превышает допустимый диапазон для типа `int`, расширяется до диапазона `unsigned int`. Если значение константы превышает и этот диапазон, то компилятор рассматривает ее как тип `long int`. Если и этого диапазона недостаточно для использования этого значения, то предпринимается попытка использовать тип `unsigned long int`. Если недостаточно и этого типа, то берется диапазон, связанный с типом `long long int`, и наконец используется тип `unsigned long long int`.

Когда объявляются переменные типа `long long int`, `long int`, `short int` или `unsigned int`, то можно пропустить ключевое слово `int`. Поэтому целочисленную переменную без знака `counter` можно объявить как

```
unsigned counter;
```

Переменные типа `char` тоже можно объявлять как беззнаковые.

Квалификатор `signed` используется для дополнительного указания компилятору на то, что задаваемая переменная может принимать значения как со знаком, так и без знака. Такие квалификаторы в основном используются при объявлении переменных типа `char`, о чем более подробно будет рассказано в главе 14, "Дополнительные сведения о типах данных".

Вас не должно смущать, что типы данных описаны несколько поверхностно. Далее в этой книге отдельные типы будут подробно рассмотрены в реальных примерах. В главе 14 даны более подробные сведения о типах данных и их преобразованиях. В табл. 4.1 перечислены все рассмотренные типы данных и их квалификаторы.

Арифметические выражения

В языке C, как и в большинстве языков программирования, знак плюс (+) используется для суммирования двух значений, знак минус (-) используется для получения разности двух значений, знак звездочка (*) – для перемножения двух значений, а обратная черта (/) – для деления двух значений. Эти операторы называются бинарными арифметическими *операторами*, поскольку они выполняют операции с двумя значениями, а членами (часто их называют *операндами*).

Вы уже ознакомились с тем, как выполняется сложение на языке программирования C, далее, в листинге 4.2, будут рассмотрены простейшие операции вычитания, перемножения и деления.

Таблица 4.1. Основные типы данных

Тип	Пример записи	Символ форматирования
char	'a', '\n'	%c
_Bool	0,1	%i, %u
short int	—	%hi, %hx, %ho
unsigned short int	—	%hu, %hx, %ho
int	12, -97, 0xFFE0, 0177	%i, %x, %o
unsigned int	12u, 100U, 0xFFu	%u, %x, %o
long int	12L, -2001, 0xffffL	%li, %lx, %lo
unsigned long int	12UL, 100ul, 0xffeeUL	%lu, %lx, %lo
long long int	0xe5e5e5e5LL, 500ll	%lli, %llx, %llo
unsigned long long int	12ull, 0xffeeULL	%llu, %llx, %llo
float	12.34f, 3.1e-5f, 0x1.5p10, 0x1P-1	%f, %e, %g, %a
double	12.34, 3.1e-5, 0x.1p3	%f, %e, %g, %a
long double	12.34l, 3.1e-5l	%Lf, %Le, %Lg

В последних двух утверждениях программы представлены последовательности операторов, в которых необходимо учитывать *приоритет*, или *старшинство*, оператора над другим оператором. Каждый оператор в языке программирования C имеет определенный приоритет, который учитывается, когда подряд записано несколько операторов и производится их последовательное выполнение. Оператор с более высшим приоритетом выполняется первым. Выражения, состоящие из операторов одинакового приоритета, выполняются последовательно слева направо или справа налево, в зависимости от оператора.

Это называется *ассоциативностью операторов*. В приложении А приведен полный список операторов, их приоритеты и правила ассоциации.

Листинг 4.2. Использование арифметических операторов

```
// Использование различных арифметических операторов
#include <stdio.h>
int main (void)
{
    int a = 100;
    int b = 2;
    int c = 25;
    int d = 4;
    int result;

    result = a - b; // Вычитание.
    printf ("a - b = %i\n", result);

    result = b * c; // Перемножение.
    printf ("b * c = %i\n", result);

    result = a / c; // Деление.
    printf ("a / c = %i\n", result);
}
```

```

result = a + b * c; // Старшинство.
printf ("a + b * c = %i\n", result);
printf ("a * b + c * d = %i\n", a * b + c * d);

return 0;
)

```

Листинг 4.2. Вывод

```

a - b = 98
b * c = 50
a / c = 4
a + b * c = 150
a * b + c * d = 300

```

После объявления целочисленных переменных `a`, `b`, `c`, `d` и `result` в программе переменной `result` присваивается результат вычитания переменной `b` из `a` и затем полученное значение отображается с помощью вызова процедуры `printf` с соответствующим символом форматирования.

В следующем утверждении

```
result = b * c;
```

производится перемножение значений переменных `b` и `c` и результат сохраняется в переменной `result`. Затем результат перемножения отображается с помощью процедуры `printf`, использующей соответствующие аргументы.

В следующем утверждении программы представлен оператор деления — косая черта (/). Полученное в результате деления значение 4 отображается с помощью процедуры `printf`, расположенной сразу за утверждением деления переменных `a` на `c`.

В большинстве компьютерных систем попытка деления на нуль завершается аварийным выходом из программы. Но даже если программа не завершается аварийно, результат, полученный в результате деления на нуль, будет абсолютно бессмысленным.

Это может произойти при использовании компилятора `gcc` в среде `Windows`. В системах `Unix` програма может завершиться аварийно, а может дать значение 0, как результат целочисленного деления на нуль, и значение "Infinity" при делении вещественных чисел.

В главе 6 вы узнаете, как произвести проверку деления на нуль до выполнения операции деления. Если выясняется, что будет производиться деление на нуль, необходимо предпринять определенные действия, чтобы избежать этого.

В результате выполнения выражения

```
a + b * c
```

будет получено значение 150, а не значение 2550 ($102 \cdot 25$). И именно значение 150 будет выведено на экран. Такое значение получается потому, что, как и в большинстве других языков программирования, в языке программирования C вычисление выражения производится слева направо, но при этом учитываются правила старшинства операторов. В данном случае операции умножения и деления имеют более высокий приоритет, чем операции сложения и вычитания. Поэтому выражение

```
a + b * c
```


необходимо рассматривать как выражение

$a + (b * c).$

В последнем случае используются обычные алгебраические правила, когда сначала вычисляется выражение в скобках.

Если вы хотите изменить порядок выполнения операторов, то можете использовать круглые скобки. Поэтому выражение, приведенное выше, вполне допустимо в языке программирования C, и в нем просто дополнительно выделяется правило использования старшинства операторов. Поэтому все утверждение можно написать следующим образом.

```
result = a + (b * c);
```

И если его подставить с листинг 4.2, то результат будет один и тот же. Однако, если написать выражение следующим образом

```
result = (a + b) * c;
```

то результирующее значение будет соответствовать 2550, поскольку сначала переменная a (100) будет сложена с переменной b (2), и только потом будет произведено умножение на переменную c (25).

Круглые скобки можно вкладывать друг в друга, в этом случае сначала будет вычисляться выражение, заключенное в "самые внутренние" скобки. Но при этом необходимо быть внимательным и ставить столько закрывающих скобок, сколько есть открывающих скобок.

Вы, наверное, обратили внимание, что в последнем утверждении программы из листинга 4.2, которое является абсолютно правильным, вычисление выражения производится непосредственно в функции `printf`, в которой вместо второго параметра подставлено само выражение. Выражение

```
a * b + c * d
```

вычисляется в соответствии с правилами старшинства, и его можно уточнить, поставив круглые скобки

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

что приведет к вычислению значений

```
(100 * 2) + (25 * 4).
```

После обработки в процедуре `printf` на экран будет выведено значение 300.

Целочисленная арифметика и оператор унарный минус

В программе из листинга 4.3 производится закрепление уже полученных знаний и вводится концепция целочисленной арифметики.

Листинг 4.3. Дополнительные примеры арифметических операторов

```
// Еще несколько арифметических выражений
#include <stdio.h>
int main (void)
{
    int    a = 25;
    int    b = 2;
    float  c = 25.0;
    float  d = 2.0;
```

```

printf ("6 + a / 5 * b = %i\n", 6 + a / 5 * b);
printf ("a / b * b = %i\n", a / b * b);
printf ("c / d * d = %f\n", c / d * d);
printf ("-a = %i\n", -a);

return 0;
}

```

Листинг 4.3. Вывод

```

6 + a / 5 * b = 16
a / b * b = 24
c / d * d = 25.000000
-a = -25

```

Дополнительные пробелы, вставленные между ключевым словом `int` и именами переменных в первых четырех утверждениях, служат для более наглядного оформления программы и позволяют сделать программу более читабельной. Вы также, наверное заметили, что во всех приведенных программах пробелами выделен каждый оператор. Это также сделано в целях более удобного зрительного восприятия. Вообще, вы можете добавлять сколь угодно пробелов там, где разрешен хотя бы один пробел. Несколько дополнительных нажатий на клавишу пробела могут в конечном итоге сделать программу более понятной.

Выражение в первой процедуре `printf` из листинга 4.3 должно вычисляться с использованием приоритетов операций. Поэтому вычисление будет происходить следующим образом.

1. Поскольку операция деления имеет более высокий приоритет, чем сложение, то сначала производится деление значения 25 на 5. В результате получим значение 5.
2. Так как операция умножения также имеет более высокий приоритет, чем сложение, то промежуточный результат 5 будет умножен на 2 (значение переменной `b`), при этом будет получен новый промежуточный результат 10.
3. Наконец производится сложение значений 6 и 10 и в результате получится окончательный ответ 16.

Конечный результат, полученный при вычислении выражения во второй процедуре `printf`, может вас удивить. Вы будете последовательно выполнять операции одного уровня приоритета, т.е. разделите `a` на `b` и затем умножьте этот промежуточный результат на `b`, что в результате должно дать значение 25.

Но после выполнения программы на экран будет выведено результирующее значение 24. Как же получилось, что в результате вычисления этого выражения компьютер потерял единичку? Все дело в том, что при вычислении этого выражения использовались правила целочисленной арифметики.

Если вы внимательнее посмотрите на объявления переменных `a` и `b`, то увидите, что обе эти переменные объявлены как целочисленный тип `int`. Если оба оператора, используемые в какой-либо операции, являются целыми числами, то в системе `C` вычисление производится по правилам целочисленной арифметики. В нашем случае будет отброшена дробная часть. Поэтому когда выполняется деление переменной `a` на `b`, т.е. деление значений 25 на 5, будет получен промежуточный результат 12, а не 12.5.

Умножение этого значения на 2 даст окончательный результат 24. Этим и объясняется потеря одной единички. Не забывайте, что когда производится деление целых чисел, в результате всегда получается целое число.

Как можно увидеть из предпоследнего утверждения `printf` в листинге 4.3, если те же самые операции производятся с использованием переменных вещественного типа вместо целочисленного, то все получается правильно.

Принятие решения о том, какие типы переменных нужно использовать, зависит того, какие задачи вы перед собой ставите. Если вы считаете, что вам не нужны дробные части, используйте целочисленные переменные. В результате программа будет выполняться быстрее. С другой стороны, если вам необходима точность, получаемая при вычислениях с вещественными числами, то используйте типы `float`, `double` или `long double`, в зависимости от требуемой точности вычислений. Точность будет зависеть как от величины чисел, так и от типа используемых переменных.

В последнем утверждении `printf` значение переменной будет отрицательным, т.к. используется оператор *унарный минус*. Этот оператор используется с единичными переменными или значениями, в противоположность бинарным операторам, которые используются с двумя переменными или значениями. При этом следует помнить, что знак минус играет двойную роль. Как бинарный оператор он используется при операциях вычитания, и как унарный оператор он используется для получения отрицательных чисел.

Унарный оператор минус имеет более высокий приоритет, чем все другие арифметические операторы, за исключением унарного плюса, который имеет тот же самый приоритет. Поэтому выражение.

```
c = -a * b;
```

представляет умножение значений двух переменных `-a` и `b`. Напомню еще раз, что в приложении А можно найти справочные материалы по всем операторам и их приоритетам.

Деление по модулю

Следующей арифметической операцией, рассматриваемой в этой главе, является операция деления по модулю, для обозначения которой используется символ процента (%). Проанализируйте программу, изображенную на листинге 4.4, и постарайтесь понять, как выполняется операция деления по модулю.

Листинг 4.4. Демонстрация деления по модулю

```
// Оператор деления по модулю
#include <stdio.h>
int main (void)
{
    int a = 25, b = 5, c = 10, d = 7;

    printf ("a %% b = %i\n", a % b);
    printf ("a %% c = %i\n", a % c);
    printf ("a %% d = %i\n", a % d);
    printf ("a / d * d + a %% d = %i\n",
           a / d * d + a % d);

    return 0;
}
```

Листинг 4.4. Вывод

```
a % b = 0  
a % c = 5  
a % d = 4  
a / d * d + a % d = 25
```

В первом утверждении процедуры `main` объявляются переменные `a`, `b`, `c`, `d` и задаются их значения.

Как вы уже знаете, в процедуре `printf` знак процента используется как специальный символ, который ставится перед символом типа и указывает на вывод значения определенного типа. Однако если за знаком процента поставить еще один знак процента, то компилятор воспримет это как указание вывести символ процента и поставить его в соответствующем месте.

Проанализировав программу, вы должны понять, что результатом деления по модулю является остаток от деления первого значения на второе. В первом случае это будет деление по модулю числа 25 на 5, в результате которого получим остаток 0. Если разделить по модулю числа 25 и 10, то остаток будет равен 5, что и является результатом этой операции, который мы видим на экране. Деление по модулю 25 на 7 даст остаток 4, что и видно в третьей строке ответа.

Последняя строка вывода для программы из листинга 4.4 требует некоторых пояснений. Во-первых, вы должны обратить внимание, что само утверждение занимает две строки. Такое написание разрешено в языке программирования C. Но при этом необходимо уточнить, что переход на следующую строку разрешен только в том месте, где может стоять пробел (исключения можно делать только при написании символьных строк, о чем будет подробнее рассказано в главе 10, "Символьные строки"). Иногда бывает не только желательно, но и необходимо располагать утверждение на нескольких строках. Продолжение утверждения на новой строке визуально выделяется отступами для лучшего восприятия всего утверждения в целом.

Обратите также внимание на вычисления выражения в последнем утверждении. Напомню, что результатом операций с целочисленными значениями является целочисленное значение.

Любой остаток, получающийся в результате деления целых чисел, будет отброшен. Поэтому разделив 25 на 7, что требуется в выражении `a/d`, получим промежуточное значение 3. Умножив это значение на значение, связанное с переменной `d`, которое равно 7, получим промежуточный результат 21. Наконец, добавляя остаток от деления по модулю `a` на `d`, получим окончательный результат 25. И то, что это значение равно значению переменной `a`, не является простым совпадением. В общем, выражение

```
a / b * b + a % b
```

всегда будет иметь значение, равное `a`, при условии, что переменные `a` и `b` объявлены как целочисленные типы.

В действительности оператор деления по модулю предназначен только для операций с целыми числами.

В шкале старшинства оператор деления по модулю имеет приоритет, равный приоритету операторов умножения и деления. Это означает, что выражение

```
table, + value % TABLE_SIZE
```

будет вычисляться как

```
table + (value % TABLE_SIZE).
```

Целочисленные и вещественные преобразования

Для эффективной разработки программ на языке C необходимо хорошо понимать правила явного преобразования целочисленных и вещественных значений. В программе из листинга 4.5 демонстрируются несложные преобразования между цифровыми типами данных. Вы также должны знать, что некоторые компиляторы могут отображать предупреждающие сообщения о том, что выполнено преобразование типов.

Листинг 4.5. Преобразования между целыми и вещественными числами

```
// Базовые преобразования в языке программирования C
#include <stdio.h>
int main (void)
{
    float f1 = 123.125, f2;
    int i1, i2 = -150;
    char c = 'a';

    i1 = f1; // Преобразование вещественного числа в целое.
    printf ("%f assigned to an int produces %i\n", f1, i1);

    f1 = i2; // Преобразование целого числа в вещественное.
    printf ("%i assigned to a float produces %f\n", i2, f1);

    f1 = i2 / 100; // Целочисленное деление.
    printf ("%i divided by 100 produces %f\n", i2, f1);

    f2 = i2 / 100.0; // Деление целого числа на вещественное.
    printf ("%i divided by 100.0 produces %f\n", i2, f2);

    f2 = (float) i2 / 100; // Оператор явного приведения типов.
    printf ("(float) %i divided by 100 produces %f\n", i2, f2);

    return 0;
}
```

Листинг 4.5. Вывод

```
123.125000 assigned to an int produces 123
-150 assigned to a float produces -150.000000
-150 divided by 100 produces -1.000000
-150 divided by 100.0 produces -1.500000
(float) -150 divided by 100 produces -1.500000
```

Всякий раз, когда вещественное значение присваивается переменной, объявленной как целое число, дробная часть вещественного значения отбрасывается. Поэтому, когда в предыдущей программе значение переменной `f1` присваивается переменной `i1`, число 123.125 усекается и только целая часть, или значение 123, сохраняется в переменной `i1`, что подтверждается первой строкой вывода.

Присваивание целочисленного значения переменной вещественного типа не вносит никаких изменений в целочисленное значение. Оно просто сохраняется в переменной вещественного типа как вещественное значение. Во второй строке программы

значение переменной `i2` (`-150`) корректно преобразовывается и сохраняется в переменной `f1`, как видно из второй строки вывода.

В следующих двух утверждениях демонстрируются правила, которые необходимо учитывать при составлении арифметических выражений. Во-первых это целочисленные арифметические операции, которые уже обсуждались в этой главе. Если в операции используются два целочисленных типа (это относится к типам `short`, `unsigned`, `long` и `long long integers`), то операция выполняется по правилам целочисленной арифметики. Поэтому дробная часть, полученная в результате выполнения арифметической операции с целыми числами, будет отброшена, даже если результат присваивается вещественной переменной. В нашем случае производится деление целочисленной переменной `i2` на целочисленную константу `100` и деление выполняется по правилам целочисленной арифметики.

В результате деления `-150` на `100` получим значение `-1`, и именно это значение будет сохранено в переменной вещественного типа `f1`.

В следующей операции деления, производимой в предыдущей программе, используются целочисленная переменная и вещественная константа. Если любой операнд, используемый в арифметической операции, является операндом вещественного типа, то операция выполняется по правилам вещественной арифметики. Поэтому когда переменная `i2` делится на `100.00`, то деление выполняется по правилам вещественной арифметики и результат будет равен `-1.5`, который и присваивается переменной вещественного типа `f1`.

Оператор приведения типов

Последняя операция деления в программе из листинга 4.5, которая представлена ниже.

```
f2 = (float) i2 / 100; // Оператор явного приведения типов.
```

использует оператор приведения типов. Оператор приведения типов `(float)` превращает тип переменной `i2` в вещественный тип, с которым в дальнейшем и производятся вычисления.

Этот оператор не изменяет саму переменную `i2` и его действие аналогично действию унарных операторов. Аналогично, поскольку выражение `-a` никак не действует на переменную `a`, выражение `(float) a` не изменяет переменную `a`.

Оператор преобразования типов имеет более высокий приоритет, чем все арифметические операторы, за исключением унарных минус и плюс. Конечно, в случае необходимости в выражении всегда можно использовать круглые скобки и выполнять операции в требуемом порядке.

Рассмотрим еще один пример приведения типов в выражении

```
(int) 29.55 + (int) 21.99
```

вычисление которого в языке программирования C будет выполнено как

```
29 + 21
```

поскольку операторы приведения типов отсекут дробные части вещественных чисел. В результате вычисления выражения

```
(float) 6 / (float) 4
```

будет получено значение 1.5 и это выражение можно написать так как показано ниже

```
(float) 6 / 4
```

что не изменит результата.

Объединение операций с присваиванием. Операторы присваивания

Язык программирования C позволяет объединять арифметические операторы с оператором присваивания, используя следующий формат.

```
op=
```

В этом формате аббревиатурой `op` обозначен любой арифметический оператор, включая `+`, `-`, `..`, `/` и `%`. В дополнение к этому, вместо `op` можно ставить любой оператор для сдвига битов и маскирования, о чем речь пойдет далее.

Рассмотрим следующее утверждение.

```
count += 10;
```

Выполнение т. н. оператора “плюс-равно” заключается в том, что вычисляется выражение с правой стороны оператора, затем промежуточное значение суммируется со значением с левой стороны оператора и наконец, полученное значение присваивается переменной с правой стороны. Поэтому приведенное выше утверждение можно заменить следующим эквивалентным утверждением.

```
count = count + 10;
```

В выражении

```
counter -= 5
```

используется оператор “минус-равно”, который присваивает переменной `counter` значение, равное разности `counter` и 5. То есть это эквивалентно следующему выражению.

```
counter = counter - 5
```

Также несложно понять и следующее выражение

```
a /= b + c
```

в котором сначала вычисляется сумма значений переменных `b` и `c`, затем на полученное значение делится переменная `a`, и наконец полученное значение присваивается переменной `a`. Суммирование выполняется в первую очередь, т.к. оператор суммирования имеет более высокий приоритет, чем оператор присваивания. В действительности все операторы, кроме точки с запятой, имеют более высокий приоритет, чем оператор присваивания.

Приведенное выше выражение идентично следующему.

```
a = a / (b + c)
```

Введение таких операторов можно объяснить тремя причинами. Во-первых, утверждения программы легче писать, т.к. не надо дублировать операнд перед знаком

равенства. Во-вторых, использование совмещенных операторов обычно лучше воспринимается при чтении. И в-третьих, использование таких операторов будет способствовать получению более быстродействующей программы, т.к. компилятор может сгенирировать более короткую последовательность кодов.

Типы `_Complex` и `_Imaginary`

Необходимо упомянуть еще о двух типах переменных, которые называются `_Complex` и `_Imaginary` и предназначены для работы с комплексными и мнимыми числами. Поддержка типов `_Complex` и `_Imaginary` не является обязательной в компиляторах языка C. Более полная информация относительно этих типов представлены в приложении A.

Во время написания книги компилятор gcc версии 3.3 не имел полной поддержки этих типов данных.

Упражнения

1. Наберите и запустите на выполнение пять программ, описанных в этой главе. Сравните данные, полученные после выполнения каждой программы, с данными, приведенными в книге.

2. Какие из следующих имен являются недопустимыми? Почему?

```
Int char 6_05
Calloc Xx alpha_beta_routine
floating_1312 z
ReInitialize _ A$
```

3. Какие из следующих констант являются недопустимыми? Почему?

```
123.456 0x10.5 0X0G1
0001 0xFFFF 123L
0Xab05 0L -597.25
123.5e2 .0001 +12
98.6F 98.7U 17777s
0996 -12E-12 07777
1234uL 1.2Fe-7 15,000
1.234L 197u 100U
0XABCDEFL 0xabcu +123
```

4. Напишите программу, которая преобразовывает величину 27 градусов по Фаренгейту (F) в градусы по Цельсию (C). Используйте следующую формулу.

$$C = (F - 32) / 1.8$$

5. Какой вывод будет сделан в следующей программе?

```
#include <stdio.h>
int main (void)
{
    char c, d;
    c = 'd';
```



```

d = c;

printf (" d = %c\n", d);
return 0;
}

```

6. Напишите программу, которая вычисляет следующий полином.

$3x^3 - 5x^2 + 6$
 для $x = 2.55$.

7. Напишите программу, которая вычисляет следующее выражение и отображает результат (для отображения результата используйте формат в виде экспоненты).

$(3.31 \times 10^{-8} \times 2.01 \times 10^{-7}) / (7.16 \times 10^{-6} + 2.01 \times 10^{-8})$

8. Для округления значения целочисленной переменной i до значения, которое будет делиться без остатка на следующее целочисленное значение j , можно использовать следующую формулу.

$\text{Next\_multiple} = i + j - i \% j$

Например, для округления 266 дней до большего значения, которое будет без остатка делиться на 7 дней, произведем расчет по данной формуле.

$\text{Next\_multiple} = 256 + 7 - 256 \% 7$
 $= 256 + 7 - 4$
 $= 259$

Напишите программу для определения значения первой переменной i , которая будет делиться без остатка на следующее целочисленное значение j .

i	j
365	7
12,258	23
996	4

Программные циклы

Если вы равномерно разместите 15 точек на поверхности треугольника, то окончательно может получиться распределение, подобное показанному на рис. 5.1.



Рис. 5.1. Распределение точек по поверхности треугольника

Первая строка содержит одну точку, вторая содержит две точки и т.д. В общем, количество точек, составляющих треугольник и имеющих n строк является суммой целых чисел от 1 до n . Такая сумма называется суммой *треугольных чисел*. Если последовательность начинается с 1, то сумма четырех треугольных чисел от 1 до 4 составляет 10 ($1 + 2 + 3 + 4 = 10$).

Предположим, что вы хотите написать программу, которая рассчитывает и отображает на экране суммы треугольных чисел. Разумеется, для небольшого числа строк можно легко рассчитать эти значения и вручную, но перед вами стоит задача написать программу на языке C для решения этой проблемы. Такая программа показана на листинге 5.1.

Программа довольно простая и хорошо подходит для расчета суммы нескольких треугольных чисел. Но если вам понадобится рассчитать сумму из двухсот чисел, то это будет довольно утомительно вводить числа от 1 до 200. К счастью, есть более легкий путь.

Листинг 5.1. Расчет треугольного числа

```
// Программа рассчитывает сумму восьми треугольных чисел.
#include <stdio.h>
int main (void)
{
    int triangularNumber;

    triangularNumber = 1 + 2 + 3 + 4 + 5 + 6 + 7 + 8;
    printf ("The eighth triangular number is %i\n",
```

```
    triangularNumber);  
    return 0;  
}
```

Листинг 5.1. Вывод

The eighth triangular number is 36

Одним из фундаментальных свойств компьютера является способность повторно выполнять последовательность из нескольких утверждений. Такие возможности выполнения циклов позволяют разрабатывать короткие программы, содержащие несколько утверждений но при этом выполняющие несколько тысяч или даже миллионов утверждений. В языке программирования С имеется три утверждения для выполнения программных циклов. Это цикл `for`, цикл `while` и цикл `do`. Каждый из этих циклов подробно будет описан в этой главе.

Цикл `for`

Рассмотрение циклов начнем с цикла `for`, для чего рассмотрим программу из листинга 5.2 в которой производится расчет суммы 200 треугольных чисел. Попробуйте сами разобраться, как работает программа с циклом `for`.

Листинг 5.2. Расчет суммы 200 треугольных чисел

```
/* Программа расчета суммы 200 треугольных чисел.  
   Введение в цикл for. */  
#include <stdio.h>  
int main (void)  
{  
    int n, triangularNumber;  
    triangularNumber = 0;  
  
    for ( n = 1; n <= 200; n = n + 1 )  
        triangularNumber = triangularNumber + n;  
    printf ("The 200th triangular number is %i\n", triangularNumber);  
  
    return 0;  
}
```

Листинг 5.2. Вывод

The 200th triangular number is 20100

Необходимо сделать несколько замечаний к программе из листинга 5.2. Способ, каким производится подсчет 200 треугольных чисел, на самом деле не отличается от способа, используемого ранее для подсчета 8-ми чисел в листинге 5.1. Просто суммируются числа от 1 до 200. Цикл `for` предоставляет механизм, который позволяет избежать явного написания каждого из 200 чисел. В некотором смысле, в этом цикле сами числа просто генерируются и нет необходимости их писать.

Общий формат утверждения для цикла `for` будет следующим:

```
for (init_expression; loop_condition; loop_expression)
program statement
```

Три выражения, заключенные в круглые скобки — `init_expression`, `loop_condition` и `loop_expression`, задают условия выполнения программного цикла.

Утверждение, которое непосредственно следует за закрывающей круглой скобкой (разумеется, оно оканчивается точкой с запятой), может быть любым допустимым утверждением языка C и является телом цикла. Это утверждение выполняется столько раз, сколько определено параметрами, установленными для цикла `for`.

Первый параметр для цикла `for`, помеченный как `init_expression`, используется для задания начального значения цикла. В программе из листинга 5.2 эта присваивание значения 1 переменной `n`. Как можно видеть, присваивание производится в соответствии с правилами языка C.

Второй компонент цикла `for` определяет условие или условия, в соответствии с которыми будет происходить выход из цикла. Другими словами, повторение будет продолжаться до тех пор, пока эти условия выполняются. Обратившись вновь к листингу 5.2, можно увидеть, что вместо параметра `loop_condition` подставлено следующее выражение отношения:

```
n <= 200
```

Это выражение необходимо понимать как “`n` меньше или равно 200”. Оператор “меньше или равно” (который записывается как символ меньше “`<`” с непосредственно за ним следующим символом равенства “`=`”) является одним из нескольких *операторов отношения*, примененных в языке программирования C. Эти операторы используются для проверки специфических условий. Результатом таких проверок может быть истина (ключевое слово `TRUE`), если условие выполняется, или ложь (ключевое слово `FALSE`), если условие не выполняется.

Операторы отношения

В табл. 5.1 перечислены все операторы отношения, используемые в языке программирования C.

Таблица 5.1. Операторы отношения

Оператор	Значение	Пример
<code>==</code>	Равно	<code>count == 10</code>
<code>!=</code>	Не равно	<code>flag != DONE</code>
<code><</code>	Меньше	<code>a < b</code>
<code><=</code>	Меньше или равно	<code>low <= high</code>
<code>></code>	Больше	<code>pointer > end_of_list</code>
<code>>=</code>	Больше или равно	<code>j >= 0</code>

Операторы отношения имеют более низкий приоритет, чем арифметические операторы. Это означает, что например, следующее выражение:

```
a < b + c
```

должно выполняться как

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

Результатом вычисления этого выражения будет TRUE, если значение операнда *a* меньше значения суммы *b+c*. В противном случае результатом будет значение FALSE.

Обратите внимание на оператор равенства "==" и не путайте его с оператором присваивания "=". Следующее выражение:

```
a == 2
```

производит проверку на равенство переменной *a* значению 2, тогда как в выражении:

```
a = 2
```

переменной *a* присваивается значение 2.

Очевидно выбор оператора присваивания зависит от условий проверки и иногда можно использовать несколько операторов для проверки одного и того же условия, что зависит от ваших предпочтений. Например, выражение:

```
n <= 200
```

полностью эквивалентно выражению:

```
n < 201
```

Вернемся к нашему примеру и посмотрим на утверждение, которое составляет тело цикла `for`:

```
triangularNumber = triangularNumber + n;
```

Это утверждение последовательно выполняется до тех пор, пока результатом вычисления выражения отношения будет значение TRUE, или в нашем случае значение переменной *n* будет меньше или равно 200. В теле цикла выполняется суммирование переменной `triangularNumber` с переменной *n*, и результат суммирования присваивается переменной `triangularNumber`.

Когда заданное условие больше не выполняется, то программа продолжает выполняться с утверждения, непосредственно следующим на циклом `for`.

В качестве последнего параметра для цикла `for` используется выражение, которое выполняется каждый раз, когда заканчивается обработка тела цикла. В программе из дистрибутива 5.2 это будет увеличение значения переменной *n* на единицу. Так как значение переменной *n* постоянно возрастает от 1 до 200, то и значение переменной `triangularNumber`, которая суммируется с переменной *n*, возрастает.

Нет ничего плохого в том, что значение переменной *n* выйдет за пределы диапазона, то есть примет значение 201. Это значение не будет суммироваться со значением переменной `triangularNumber`, поскольку выполнение тела цикла прекратится до того, как только условие перестанет выполняться, т.е. переменная *n* примет значение 201.

Вычисление цикла `for` производится по следующему сценарию, представленному как последовательность шагов:

1. Сначала вычисляется выражение инициализации. В результате вычисления обычно присваивается значение переменной, которая будет использоваться внутри цикла. На такую переменную ссылаются как на индексную переменную и начальное значение в большинстве случаев будет 0 или 1.

2. Затем проверяется выражение условия для продолжения цикла. Если условие не выполняется (результатом вычисления является значение FALSE), то цикл немедленно заканчивается. В противном случае выполнение цикла продолжается с утверждения, которое следует за заголовком цикла.
3. Выполняются все утверждения, составляющие тело цикла.
4. Вычисляется выражение состояния цикла. Это выражение обычно используется для изменения индексной переменной. В большинстве случаев к ней добавляется или вычитается единица.
5. Возврат к шагу 2.

Запомните, что выражение условия вычисляется до начала вычисления тела цикла, поэтому тело цикла может не выполниться ни разу. Также запомните, что нельзя ставить точку с запятой за закрывающей скобкой заголовка цикла, т.к. это будет считаться концом цикла.

Поскольку в программе из листинга 5.2 действительно генерируются все 200 треугольных чисел при выполнении цикла, возможно будет интересно создать таблицу этих чисел. Но не будет занимать место и распечатаем только таблицу из 10 треугольных чисел.

Приведенная ниже программа (листинг 5.3) выполняет именно эту задачу.

Листинг 5.3. Генерация таблицы треугольных чисел

```
// Генерация таблицы треугольных чисел.
#include <stdio.h>
int main (void)
{
    int n, triangularNumber;

    printf ("TABLE OF TRIANGULAR NUMBERS\n\n");
    printf (" n   Sum from 1 to n\n\n");
    printf ("--- -----\n\n");
    triangularNumber = 0;
    for ( n = 1; n <= 10; ++n ) {
        triangularNumber += n;
        printf (" %i          %i\n", n, triangularNumber);
    }
    return 0;
}
```

Листинг 5.3. Вывод

TABLE OF TRIANGULAR NUMBERS

n	Sum from 1 to n
--- ----	
1	1
2	3
3	6
4	10
5	15
6	21
7	28

8	36
9	45
10	55

Всегда старайтесь добавить несколько дополнительных утверждений `printf` в программу для того, чтобы сделать вывод более информативным. В листинге 5.3 этой цели служат три первых утверждения, с помощью которых печатается заголовок таблицы и необходимые пояснения.

Обратите внимание, что в первом утверждении `printf` содержится два символа новой строки. Это используется не только для перевода вывода на новую строку, но и для того, чтобы вставить дополнительную пустую строку при выводе на экран.

После того как заголовок будет выведен, в программе начнется вычисление первых 10 треугольных чисел. Переменная `n` используется для подсчета текущего числа и расчета треугольного числа, которое сохраняется в переменной `triangularNumber`.

Выполнение цикла `for` начинается с установки значения переменной `n` в 1. Вспомните, что утверждение, непосредственно следующее за заголовком цикла `for`, составляет тело цикла. Но как быть, если вы хотите выполнять в теле цикла не одно утверждение, а несколько? Это можно сделать, заключив такие утверждения в фигурные скобки. Система будет трактовать такую группу утверждений, или *блок*, как один элемент. В общем случае, в любом месте, где можно поставить одно утверждение, можно использовать и блок. При этом не забывайте выделять блок парой фигурных скобок.

Поэтому в программе из листинга 5.3 тело цикла составляют два утверждения, в которых производится вычисление треугольного числа, сохраняемого в переменной `triangularNumber`, и вывод на экран (утверждения `printf`), непосредственно следующие за заголовком цикла `for`. Обратите внимание на то, как эти утверждения выделены в программе — они имеют дополнительный отступ, что позволяет их отличать визуально. Также хочу обратить ваше внимание на то, что программисты используют различные стили написания утверждений. Некоторые предпочитают писать несколько по другому, чем с листинге 5.3, и делают это следующим образом:

```
for ( n = 1; n <= 10; ++n )
{
    triangularNumber += n;
    printf ( " %i %i\n", n, triangularNumber);
}
```

То есть открывающаяся фигурная скобка помещается на следующей строке после заголовка цикла `for`. Это дело вкуса и не влияет на выполнение программы.

Следующее треугольное число вычисляется простым суммированием текущего значения переменной `n` с предыдущим треугольным числом. В этом случае удобно использовать оператор “плюс равно”, о котором говорилось в главе 4 “Переменные, типы данных и арифметические выражения”. Вспомните, что выражение:

```
triangularNumber += n;
```

эквивалентно выражению:

```
triangularNumber = triangularNumber + n;
```

В первый момент, когда начинается выполнение цикла `for`, “предыдущее” треугольное число имеет значение 0, поэтому значение переменной `triangularNumber`

при `n` равном 1 будет равно 1. Затем значения переменных `n` и `triangularNumber` будут отображены на экране, при этом будут вставлены пробелы в соответствии со строкой форматирования и будет сформирована строка со значениями в соответствующей колонке.

После того, как тело цикла будет выполнено, будет вычисляться выражение, определяющее состояние цикла. В нашем случае это выражение для цикла `for` выглядит несколько странно. Это похоже на типографскую ошибку, так как должно стоять выражение:

```
n = n + 1
```

а вместо этого мы видим что-то необычное:

```
++n
```

На самом деле в этом не ничего необычного и такое выражение является действительно допустимым выражением языка программирования C. Два плюса подряд обозначают *оператор инкремента*, т.е. увеличения значения операнда на единицу.

Поскольку операция увеличения на единицу довольно часто встречается в алгоритмах, то для этой операции и был выделен специальный оператор. Поэтому выражение `++n` эквивалентно выражению `n=n+1`. Хотя на первый взгляд может показаться, что использование записи `n=n+1` более наглядно, вы скоро привыкните к оператору инкремента и оцените удобство его использования.

Разумеется, ни один язык программирования, в котором используется оператор инкремента, не может обойтись без соответствующего оператора декремента, т.е. уменьшения на единицу. Такой оператор называется *оператором декремента* и записывается как два знака минус подряд. Поэтому выражение на языке C:

```
bean_counter = bean_counter - 1
```

может быть записано с использованием оператора декремента:

```
--bean_counter
```

Некоторые программисты предпочитают записывать символы `++` или `--` после операнда, например `bean_counter--`. Это также допустимо и в данном случае зависит от предпочтений программиста.

Выравнивание вывода

Если вы внимательнее присмотритесь к тому, что будет выведено на экран в результате работы программы из листинга 5.3, то можете заметить, что выводимые столбцы будут не совсем ровные, и треугольное число за номером 10 будет смещено относительно предыдущего треугольного числа. Это происходит потому, что число 10 занимает две позиции вместо одной, которую занимают все предыдущие числа от 1 до 9. Поэтому число 55 и будет дополнительно смещено на одну позицию вправо. Это небольшое искажение можно легко исправить, если соответствующее утверждение `printf` из листинга 5.3 записать следующим образом:

```
printf ("%2i          %i\n", n, triangularNumber);
```

Для того, чтобы убедиться, что теперь все будет правильно, выполните программу с доработанным утверждением `printf` и получите следующий вывод (назовем его листинг 5.3А).

TABLE OF TRIANGULAR NUMBERS

n	Sum from 1 to n
1	1
2	3
3	6
4	10
5	15
6	21
7	28
8	36
9	45
10	55

Основное изменение, которое было сделано в утверждении `printf`, это то, что был добавлен *спецификатор ширины поля*. Символы `%2i` указывают утверждению `printf` на то, что вы не только хотите отобразить целочисленное значение, но также хотите разместить это значение в двух столбцах. Любое целое число, которое будет занимать меньше, чем два столбца (это числа от 0 до 9), будет отображаться с дополнительным пробелом впереди. Это называется *выравниванием по правому разряду*.

Таким образом, использование спецификатора ширины поля гарантирует, что по крайней мере два столбца будут использоваться при отображении значения переменной `n` и вы будете уверены, что выводимые значения для переменной `triangular-number` будут выровнены по столбцам.

Если значение, которое должно быть отображено, для своего вывода требует больше столбцов, чем указано в спецификаторе ширины поля, то спецификатор ширины поля просто игнорируется и значение выводится на столько столбцах, сколько для этого требуется.

Спецификатор ширины поля можно использовать не только для отображения целочисленных значений. Соответствующие примеры программ скоро будут представлены.

Ввод данных

В программе из листинга 5.2 производится подсчет 200 треугольных чисел, и ничего более. Если вы захотите подсчитать треугольное число для 50 или 100 чисел, то вы должны будете внести изменения в программу и установить в цикле `for` соответствующие значения. Также необходимо будет изменить и сообщения, выводимые на экран.

Но ваша программа будет гораздо удобнее в работе, если вы сделаете так, чтобы она производила запрос входных данных и сама готовила необходимый вывод. При этом после ввода необходимых значений, программа произведет расчет треугольного числа именно для того значения, которое вы ввели. Такой ввод легко можно сделать, и для этого в языке программирования C предусмотрена подпрограмма с именем `scanf`. Подпрограмма `scanf` подобна функции `printf`. Отличие заключается в том, что функция `printf` используется для вывода значений из программы на терминал, а функция `scanf` предназначена для ввода значений с терминала в программу. В листинге 5.4 приведен пример программы, в которой производится запрос значения, для которого должно быть рассчитано треугольное число, расчет этого числа и вывод результата на экран.

Листинг 5.4. Запрос к пользователю на ввод значения

```
#include <stdio.h>
int main (void)
{
    int n, number, triangularNumber;

    printf ("What triangular number do you want? ");
    scanf ("%i", &number);
    triangularNumber = 0;
    for ( n = 1; n <= number; ++n )
        triangularNumber += n;
    printf ("Triangular number %i is %i\n", number, triangularNumber);

    return 0;
}
```

Текст, отображаемый на терминале, будет состоять из запроса, значения, введенного пользователем, которое будет выделено жирным шрифтом, и сообщения о результате расчета.

Листинг 5.4. Вывод

```
What triangular number do you want? 100
Triangular number 100 is 5050
```

Из этого текста видно, что число 100 было введено пользователем. Программа выполнила расчет и отображала результирующее значение 5050, которое является треугольным числом для 100 чисел. Если вы хотите произвести расчет для 10, или 30 чисел, то необходимо только ввести необходимое значение и получить результат.

Первое утверждение `printf`, которое находится в листинге 5.4 использовано для вывода запроса на ввод значения, которое пользователь должен ввести. Это всегда удобно напомнить пользователю, что он должен делать. После того, как запрос выведен, вызывается подпрограмма `scanf`. Первым аргументом при этом вызове является строка форматирования, которая напоминает строку форматирования для функции `printf`. Только в этом случае строка форматирования служит не для формирования вывода значений на экран, а для передачи в программу информации о том, какое значение должно быть введено с терминала. Как и в функции `printf`, символы `"%i"` определяют целочисленное значение.

Второй аргумент подпрограммы `scanf` является идентификатором, который будет указывать на то место в памяти, где будет сохранено значение, введенное пользователем. В этом случае необходимо использовать символ `"&"` перед именем переменной `number`. Пока не будет обсуждать в деталях необходимость использования символа `"&"`, это будет сделано в главе 11 "Указатели", где будет подробно рассказано о том, что символ `"&"` на самом деле является оператором. Пока же запомните, что этот символ всегда необходимо помещать перед именем переменной в функции `scanf`. Если вы забудете это сделать, то будет получен непредсказуемый результат или ваша программа закончится аварийно.

Продолжая обсуждение программы из листинга 5.4 еще раз обратим внимание на то, что функция `scanf` должна считать целочисленное значение с терминала и сохранить его в переменной `number`. Это значение и будет представлять треугольное число, которое пользователь хочет рассчитать.

После того, как значение будет набрано на терминале и будет сделан ввод (на клавиатуре будет нажата клавиша <Enter> или <Return>, что является сигналом окончания ввода), программа начнет выполнять расчет треугольного числа. Это происходит точно так, как и в программе из листинга 5.2, за одним различием – вместо числа 200 поставлена переменная `number`.

После расчета необходимого треугольного числа, результат будет отображен на дисплее и выполнение программы будет закончено.

Вложение циклов `for`

Программа из листинга 5.4 дает пользователю необходимую гибкость для расчета любого треугольного числа. Однако, если необходимо рассчитать несколько треугольных чисел подряд, придется запускать программу столько раз, сколько треугольных чисел необходимо рассчитать, каждый раз набирая очередное треугольное число.

Можно выполнить ту же задачу и по другому, используя более интересный способ, который можно выполнить с помощью более развитых средств языка программирования C и который позволяет облегчить выполнение задачи. Например, можно повторять программу столько раз, сколько данных необходимо вводить. Вы уже знаете, что для этих целей можно использовать цикл `for`. В программе (листинг 5.5) показано, как можно использовать два цикла `for` в одной программе.

Листинг 5.5. Использование вложенных циклов `for`

```
#include <stdio.h>
int main (void)
{
    int n, number, triangularNumber, counter;

    for ( counter = 1; counter <= 5; ++counter ) {
        printf ("What triangular number do you want? ");
        scanf ("%i", &number);
        triangularNumber = 0;
        for ( n = 1; n <= number; ++n )
            triangularNumber += n;
        printf("Triangular number %i is %i\n\n", number, triangularNumber);
    }
    return 0;
}
```

Листинг 5.5. Вывод

```
What triangular number do you want? 12
Triangular number 12 is 78

What triangular number do you want? 25
Triangular number 25 is 325

What triangular number do you want? 50
Triangular number 50 is 1275

What triangular number do you want? 75
Triangular number 75 is 2850

What triangular number do you want? 83
Triangular number 83 is 3486
```

Программа состоит из двух циклов `for`. Внешний цикл `for`

```
for(counter = 1; counter <= 5; ++counter)
```

Определяет цикл программы, который выполняется точно пять раз. Это видно из того, что переменная `counter` инициализируется значением 1 и инкрементируется до тех пор, пока оно не будет превышать значение 5 (другими словами, пока оно достигнет значения 6).

В отличие от других программ, переменная `counter` нигде в программе не используется. Единственная ее функция — это счетчик для цикла `for`.

Тем не менее, поскольку она является переменной, она должна быть объявлена в программе. Цикл программы будет состоять из всех остальных утверждений, которые заключены в фигурные скобки. Возможно, вы сможете лучше понять программу, если запишите ее узловыми моментами обычными словами.

Повторить 5 раз

```
{
    Получить значение от пользователя.
    Рассчитать требуемое треугольное число.
    Отобразить результат.
}
```

Фрагмент программы, который скрывается за словами “Рассчитать требуемое треугольное число” на самом деле состоит из утверждения, в котором переменной `triangularNumber` присваивается значение 0, и цикла `for`, который оказывается вложенным в другой цикл `for`. Это разрешено в языке программирования C и вложенные циклы могут достигать 127 уровней!

При этом, чтобы разобраться в структуре запутанной программы, такой как вложенные циклы, необходимо грамотно использовать отступы. При удачно расставленных отступах можно легко выделить каждый вложенный цикл. (Для того, чтобы сравнить, как трудно понимать программу, в которой плохо выполнено форматирование, смотри упражнение 5 в конце этой главы).

Варианты записи цикла `for`

Некоторые синтаксические варианты разрешены при формировании цикла `for`. При записи цикла `for` вы можете использовать более одной переменной, которую вы будете инициализировать перед началом выполнения цикла `for`, и применить более одного выражения, которые будут изменять эти переменные в процессе выполнения цикла.

Несколько выражений

Вы можете включить несколько выражений в каждое поле цикла `for` при условии, что в конце каждой группы будет стоять точка с запятой. Например, цикл `for`, заголовок которого записан следующим образом

```
for ( i = 0, j = 0; i < 10; ++i )
...

```

значение переменной `i` устанавливается в 0, и значение переменной `j` также устанавливается в 0 до начала выполнения цикла. Два выражения, `i=0` и `j=0`, отделяются от других выражений точкой с запятой и оба выражения рассматриваются как часть поля `init_expression` для цикла. Еще один пример цикла `for` может начинаться следующим образом.

```
for ( i = 0, j = 100; i < 10; ++i, j = j - 10 )
...

```

Здесь устанавливаются две индексные переменные *i* и *j*, которые инициализируются значениями 0 и 100 до начала выполнения цикла. Всякий раз, после выполнения тела цикла, значение переменной *i* инкрементируется на 1, а значение переменной *j* декрементируется на 10.

Пропуск полей

Подобно тому, как возникает необходимость использовать более одного выражения в отдельном поле цикла *for*, возникает необходимость исключать одно или более полей из заголовка цикла. Это легко сделать, просто пропустив определенное поле и поставив вместо него точку с запятой. Наиболее общей причиной такого пропуска в заголовке цикла *for*, является отсутствие необходимости инициализировать начальные значения, которые, соответственно, не должны изменяться. В этом случае поле *init_expression* будет просто пропущено и вместо него будет поставлена точка с запятой.

```
for ( ; j != 100; ++j )
...

```

В этом цикле используется переменная *j*, которая должна быть уже инициализирована до того, как в программе будет произведен переход к циклу.

Цикл *for*, в котором пропущено поле *looping_condition*, можно эффективно использовать как бесконечный цикл, т.е. цикл, который теоретически может выполняться вечно. Выход из таких циклов может производиться с помощью других средств, таких как утверждения *return*, *break* или *goto*, которые будут обсуждаться далее в этой книге.

Объявления переменных

Переменные можно объявлять и в поле инициализации цикла *for*. Это можно сделать так же, как переменные объявляются в начале программы. Например, в следующий цикл *for* включена целочисленная переменная *counter*, которая объявлена и инициализирована значением 1 непосредственно в цикле.

```
for ( int counter = 1; counter <= 5; ++counter )

```

Переменная *counter* известна только в пределах цикла *for* (такие переменные называются *локальными переменными*) и к ней нельзя обратиться за пределами цикла. В следующем примере цикла *for*

```
for ( int n = 1, triangularNumber = 0; n <= 200; ++n )
    triangularNumber += n;

```

объявлены две целочисленные переменные и им присвоены значения.

Цикл while

Цикл *while* еще более расширяет возможности языка программирования С относительно повторения группы утверждений. Синтаксис этой часто используемой конструкции можно представить следующим образом.

```
while ( expression )
    program statement

```

Производится расчет выражения `expression`, заключенного в круглые скобки. Если в результате расчета выражения `expression` получается результат `TRUE`, то выполняется утверждение `program statement`, следующее непосредственно за закрывающей скобкой. После выполнения этого утверждения (или группы утверждений, заключенных в фигурные скобки), вновь рассчитывается выражение `expression`. Если результатом расчета опять будет `TRUE`, то вновь будут выполнены утверждения `program statement`. Цикл повторяется до тех пор, пока в результате расчета выражения `expression` не будет получено значение `FALSE`, которое является признаком окончания цикла, после чего выполнение программы продолжается с утверждения, непосредственно следующего за утверждением `program statement`.

В примере использования цикла `while` (листинг 5.6) повторение утверждений, изменяющих и выводящих значение переменной `count` на экран, производится 5 раз.

Листинг 5.6. Знакомство с циклом `while`

```
// Программа знакомит с использованием цикла while.
#include <stdio.h>
int main (void)
{
    int count = 1;

    while ( count <= 5 ) {
        printf ("%i\n", count);
        ++count;
    }

    return 0;
}
```

Листинг 5.6. Вывод

```
1
2
3
4
5
```

Сначала в программе значение переменной `count` устанавливается равным 1. Затем начинается выполнение цикла `while`. Поскольку значение переменной `count` меньше 5, то выполняются утверждения, непосредственно следующие за заголовком цикла. Фигурные скобки служат для выделения тела цикла, которое включает утверждение `printf` и утверждение инкремента переменной `count`. После того как будет выполнен вывод на экран, можно убедиться, что цикл был выполнен точно 5 раз, пока значение переменной `count` не достигло 6.

Анализ приведенной выше программы позволяет сделать вывод, что эту же самую задачу можно выполнить и с помощью цикла `for`. В действительности всегда можно вместо цикла `for` использовать цикл `while` и наоборот. Например, наиболее общая форма цикла `for`

```
for ( init_expression; loop_condition; loop_expression )
    program statement
```

может быть заменена эквивалентным циклом `while`.

```
while ( loop_condition ) {
    program statement
    loop_expression;
}
```

После того как вы лучше познакомитесь с циклами `while` и `for`, вы сможете почувствовать разницу между ними и понять, когда логически более подходит цикл `while`, а когда цикл `for`.

В общем случае, если цикл должен выполняться заранее заданное число раз, то наилучшим выбором будет цикл `for`. Если выражения инициализации, повторения цикла и условия используют одну и ту же переменную, наилучшим выбором также будет цикл `for`.

В следующей программе приводится еще один пример использования цикла `while`. В программе рассчитывается *наибольший общий делитель* двух целых чисел. Наибольшим общим делителем (greatest common divisor – gcd) двух целочисленных значений является число, которое является наибольшим из чисел, на которые можно разделить целочисленные значения без остатка.

Например, наибольшим общим делителем для чисел 10 и 15 будет число 5, поскольку это наибольшее число, на которое можно разделить 10 и 15 без остатка.

Для получения наибольшего общего делителя для двух целых чисел можно использовать алгоритм, разработанный Евклидом еще за 300 лет до нашей эры, который можно описать следующим образом.

Задача. Найти наибольший общий делитель для двух положительных целых чисел u и v .

Шаг 1. Если v равно 0, то решение получено и наибольший общий делитель равен u .

Шаг 2. Рассчитать выражения $temp = u \% v$, $u = v$, $v = temp$ и перейти к шагу 1.

В данный момент не надо глубоко анализировать работу этого алгоритма, просто примите его на веру. Давайте лучше сосредоточимся на разработке программы для определения наибольшего общего делителя, которая будет использовать описанный алгоритм.

После того, как решение задачи определения наибольшего общего делителя будет выражено в виде алгоритма, намного упростится задача по созданию соответствующей программы для компьютера. Из анализа шагов алгоритма видно, что шаг 2 постоянно повторяется до тех пор, пока значение переменной v не станет равным 0. Поэтому при реализации алгоритма на языке программирования C удобно использовать цикл `while`.

В программе из листинга 5.7 определяется наибольший общий делитель для двух целых положительных чисел, введенных пользователем.

Листинг 5.7. Определение наибольшего общего делителя

```
/* Программа Находит наибольший общий делитель
   для двух целых положительных чисел */
#include <stdio.h>
int main (void)
{
    int u, v, temp;

    printf ("Please type in two nonnegative integers.\n");
    scanf ("%i%i", &u, &v);
```

```

while ( v != 0 ) {
    temp = u % v;
    u = v;
    v = temp;
}
printf ("Their greatest common divisor is %i\n", u);

return 0;
}

```

Листинг 5.7. Вывод

Please type in two nonnegative integers.

150 35

Their greatest common divisor is 5

Листинг 5.7. Вывод (Повторение)

Please type in two nonnegative integers.

1026 405

Their greatest common divisor is 27

Два символа “%i” в утверждении scanf свидетельствуют о том, что с клавиатуры должно быть введено два целочисленных значения. Первое значение, которое будет введено, сохраняется в переменной u типа int, а второе значение сохраняется в переменной v. Когда значения переменных вводятся с терминала, они должны отделяться друг от друга одним (или более) пробелом или переводом каретки.

После того как значения с помощью клавиатуры вводятся и сохраняются в переменных u и v, в программе начинает выполняться цикл while для расчета наибольшего общего делителя. После завершения выполнения цикла while, значение переменной u, которое представляет наибольший общий делитель, будет отображено на терминале с соответствующим сообщением.

В программе из листинга 5.8 показан еще один пример использования цикла while, с помощью которого производится вывод цифр, составляющих введенное число, на экран в обратном порядке. Например, если пользователь наберет число 1234, то программа произведет реверс цифр данного числа и выведет результат 4321.

Для того чтобы написать такую программу, сначала необходимо составить алгоритм решения данной задачи. Обычно при анализе методов решения задачи производится разработка алгоритмов. Для реверса цифр числа можно последовательно считывать отдельные цифры числа справа налево. Поэтому необходимо разработать программу, которая последовательно считывает цифры числа справа налево и выделяет каждую считанную цифру начиная с самой правой цифры. Выделенные цифры будут последовательно отображаться на дисплее, что в конечном итоге и создаст последовательность цифр в обратном порядке.

Выделение каждой правой цифры числа может быть сделано с помощью остатка от деления данного числа на 10. Например, если произвести деление по модулю числа 1234 на 10 ($1234 \% 10$), то получим число 4, которое представляет крайнюю правую цифру числа 1234 и является первой цифрой реверсированного числа. Вспомните, что деление по модулю дает остаток, полученный от деления двух целых чисел, что в нашем случае как очень кстати. Для получения следующей цифры необходимо использовать ту же самую процедуру деления по модулю числа на 10. Для получения последующего

числа необходимо просто произвести целочисленное деление числа на 10. Например, целочисленное деление 1234 на 10 ($1234/10$) даст в результате число 123. Разделив по модулю 123 на 10, получим цифру 3, т.е. очередную цифру реверсированного числа.

Этот процесс должен продолжаться до тех пор, пока все цифры исходного числа не будут выделены. Последняя цифра будет получена, когда в результате целочисленного деления на 10 будет получен 0.

Листинг 5.8. Реверс цифр, составляющих число

```
// Программа переставляет цифры числа в обратном порядке.
#include <stdio.h>
int main (void)
{
    int number, right_digit;

    printf ("Enter your number.\n");
    scanf ("%i", &number);
    while ( number != 0 ) {
        right_digit = number % 10;
        printf ("%i", right_digit);
        number = number / 10;
    }
    printf ("\n");

    return 0;
}
```

Листинг 5.8. Вывод

```
Enter your number.
13579
97531
```

Каждая цифра, выделенная из исходного числа, будет отображаться на дисплее. Обратите внимание, что в утверждение `printf`, которое находится в цикле `while`, не включен символ перевода на новую строку. Это приводит к тому, что все цифры будут отображаться последовательно на одной строке. Последнее утверждение `printf` содержит только символ новой строки, что приводит к переводу курсора в начало очередной строки.

Цикл `do`

Оба утверждения цикла, которые уже обсуждались ранее, производят проверку условия выполнения цикла до начала выполнения тела цикла. Поэтому тело цикла может ни разу не выполниться, если с самого начала результатом расчета условия выполнения цикла будет значение `FALSE`. При разработке программ часто необходимо делать проверку условия выполнения цикла после выполнения тела цикла, а не перед началом цикла. Естественно, такой цикл в языке программирования C существует и он обозначается как цикл `do`. Синтаксис цикла `do` пишет следующий вид.

```
do
program statement
while ( loop_expression );
```

Выполнение цикла `do` происходит следующим образом: сначала выполняется утверждение `program statement`, затем производится проверка условия выполнения цикла `loop_expression`. Если результатом проверки будет значение `TRUE`, выполнение цикла продолжается и утверждение `program statement` выполняется вновь. Повторение цикла будет продолжаться до тех пор, пока в результате проверки условия выполнения цикла `loop_expression` будет получаться значение `TRUE`. Когда в результате проверки условия будет вычислено значение `FALSE`, выполнение цикла прекращается и происходит переход к утверждению, непосредственно следующему за циклом.

Цикл `do` похож на цикл `while`, в котором проверка условия производится не в начале цикла, а в конце. При этом необходимо четко представлять, что, в отличие от циклов `for` и `while`, цикл `do` гарантированно выполнится хотя бы один раз.

В программе из листинга 5.8 для реверса цифр числа использовался цикл `while`. Вернитесь к этой программе и проанализируйте, что произойдет, если вы введете число 0 вместо 13579. Тогда цикл `while` не будет выполнен ни разу и на экране будет получена только пустая строка как результат выполнения второго утверждения `printf`, в котором производится вывод символа новой строки. Если же вместо цикла `while` использовать цикл `do`, то можно быть уверенным, что цикл выполнится по крайней мере один раз, при этом будет отображена по крайней мере одна цифра. В листинге 5.9 приведена скорректированная программа реверса цифр числа.

Листинг 5.9. Реализация улучшенной программы реверса цифр числа

```
// Программа переставляет в обратном порядке цифры числа.
#include <stdio.h>
int main ()
{
    int number, right_digit;

    printf ("Enter your number.\n");
    scanf ("%i", &number);
    do {
        right_digit = number % 10;
        printf ("%i", right_digit);
        number = number / 10;
    }
    while ( number != 0 );
    printf ("\n");

    return 0;
}
```

Листинг 5.9. Вывод

```
Enter your number.
13579
97531
```

Листинг 5.9. Вывод (Повторение)

```
Enter your number.
0
0
```

Согласно выводимым программой данным, при введении числа 0 программа корректно отображает цифру 0.

Утверждение break

При определенных условиях возникает необходимость прекратить выполнение цикла (например, если непредвиденно закончились входные данные или зафиксирована предпосылка аварийной ситуации). В таких случаях удобно использовать утверждение `break`.

Выполнение утверждения `break` приводит к немедленному выходу из цикла, в котором оно встретилось. Это могут быть циклы `for`, `while` или `do`. Все последующие утверждения цикла пропускаются и выполнение программы продолжается с утверждения, непосредственно следующего за циклом.

Если утверждение `break` встречается во вложенном цикле, то прекращается выполнение только того цикла, в котором это утверждение встретилось. Утверждение `break` записывается как ключевое слово `break` с последующей точкой с запятой.

```
break;
```

Утверждение continue

Утверждение `continue` действует подобно утверждению `break`, за исключением того, что оно не приводит к выходу из цикла. Вместо этого, как подсказывает само название “`continue`” (продолжение), продолжается выполнение цикла. При этом все утверждения, которые следуют за ключевым словом `continue`, автоматически пропускаются. Выполнение цикла продолжается с начала цикла в обычном режиме.

Утверждение `continue` наиболее часто используется для пропуска группы утверждений в зависимости от условий, возникающих внутри цикла, но, в любом случае, выполнение цикла продолжается. Формат утверждения `continue` пишет следующий вид.

```
continue;
```

Старайтесь не использовать утверждения `break` или `continue` до тех пор, пока вы не почувствуете все нюансы работы циклов. Эти утверждения часто приводят к ошибкам, которые впоследствии будет трудно обнаружить при отладке программы.

Теперь, когда вы уже познакомились с циклами и некоторыми другими основными конструкциями языка программирования C, мы перейдем к изучению языковых конструкций, которые позволят вам делать логические заключения и принимать решения при выполнении программы. О принятии решений подробно рассказывается в главе 6, “Принятие решений”. Но сначала постарайтесь выполнить все программы, описанные в данной главе, чтобы как можно лучше понять работу циклов.

Упражнения

1. Наберите и запустите все программы, представленные в этой главе. Сравните выводы, сделанные вашими программами, с выводами, представленными после листингов программ.
2. Напишите программу, генерирующую и отображающую таблицу чисел n и n^2 для целых значений n от 1 до 10. Постарайтесь придать выводимым строкам аккуратный вид.

3. Треугольное число можно вычислить с помощью следующей формулы:

$$\text{triangularNumber} = n (n + 1) / 2$$

для любого целочисленного значения n . Например, для числа 10 треугольным числом будет 55, что легко можно проверить, подставив это значение в приведенную формулу. Напишите программу, которая производит расчет треугольных чисел с помощью предложенной формулы.

Программа должна вычислить каждое пятое треугольное число между 5 и 50 (это числа 5, 10, 15, ..., 50).

4. Факториал целого числа n записывается как $n!$ и представляет собой последовательное произведение чисел от 1 до n . Например, факториал числа 5 рассчитывается так:

$$5! = 5 \times 4 \times 3 \times 2 \times 1 = 120$$

Напишите программу, рассчитывающую и создающую таблицу для первых 10 факториалов.

5. Приведенная ниже корректно работающая программа написана без соблюдения правил форматирования. Как можно увидеть, программа недостаточно легко читается. Ее можно сделать вообще практически нечитабельной, при этом она будет правильно компилироваться и хорошо работать. Используя программы, приведенные в данной главе в качестве примера, переформатируйте программу таким образом, чтобы она стала максимально доступной для понимания. Введите программу в компьютер и запустите ее.

```
#include <stdio.h>
int main(void){
    int n,two_to_the_n;
    printf("TABLE OF POWERS OF TWO\n\n");
    printf(" n 2 to the n\n");
    printf("---\n");
    two_to_the_n=1;
    for(n=0;n<=10;++n){
        printf("%2i %i\n",n,two_to_the_n); two_to_the_n*=2;
    }
    return 0;
}
```

6. Знак минус, помещенный в начале спецификатора ширины, приводит в отображению значений, выровненных по левому краю. Замените соответствующее утверждение `printf` в программе из листинга 5.2 приведенным ниже утверждением. Запустите программу выводом и сравните вывод, сделанный скорректированной программой, с выводом, сделанным ранее.

```
printf ("%2i %i\n", n, triangularNumber);
```

7. Десятичная точка, помещенная в начале спецификатора ширины в утверждении `printf`, имеет специальное назначение. Попытайтесь определить это назначение, запуская и анализируя следующую программу. Вводите различные значения при каждом запуске программы.

```
#include <stdio.h>
int main (void)
```

```

{
    int dollars, cents, count;

    for ( count = 1; count <= 10; ++count ) {
        printf ("Enter dollars: ");
        scanf ("%i", &dollars);
        printf ("Enter cents: ");
        scanf ("%i", &cents);
        printf ("${i}%.2i\n\n", dollars, cents);
    }
    return 0;
}

```

8. Программа из листинга 5.5 позволяет пользователю ввести только пять различных значений. Измените программу таким образом, чтобы пользователь мог вводить столько значений, сколько ему необходимо.
9. Перепишите программы из листингов 5.2 – 5.5, заменяя все циклы `for` эквивалентными циклами `while`. Запустите каждую программу и проверьте идентичность версий.
10. Что произойдет, если в программе из листинга 5.8 ввести отрицательное число? Проверьте это на практике.
11. Напишите программу, которая вычисляет сумму цифр целого числа. Например, суммой цифр целого числа 2155 будет $2 + 1 + 5 + 5$, или 13. Программа должна позволять пользователю вводить любое целое число.

6

Принятие решений

В главе 5, “Программные циклы”, вы узнали, что одним из фундаментальных свойств компьютера является способность повторно выполнять последовательность утверждений. Но есть еще одно фундаментальное свойство, о котором уже упоминалось, — это способность принимать решения. Вы наверняка обратили внимание, как способность по принятию решений была использована в различных циклах, когда необходимо было произвести выход из цикла. Без такой возможности вы бы никогда не смогли закончить цикл и последовательность программных утверждений повторялась бы снова и снова, теоретически вечно (такие программные циклы называются *бесконечными циклами*).

В языке программирования C используется несколько конструкций для принятия решений, которые будут рассмотрены в данной главе:

- оператор `if`;
- оператор `switch`;
- Условный оператор

Оператор `if`

В языке программирования C среди общих языковых конструкций для принятия решений наиболее часто используется конструкция, известная как *оператор if*. Общий формат этого утверждения представлен ниже.

```
if ( expression )  
program statement
```

Предположим, что вам необходимо выразить на языке программирования C следующее предложение “Если нет дождя, то я должен идти купаться”. Используя формат утверждения `if`, это можно сделать следующим образом.

```
if (нет дождя)  
я должен идти купаться
```

В операторе `if` используется результат вычисления выражения условия, заключенного в круглые скобки, на основе которого и принимается решение — я пойду купаться, если не будет дождя. Таким образом, в следующих утверждениях на языке C:

```
if ( count > COUNT_LIMIT )
printf ("Count limit exceeded\n");
```

Утверждение `printf` выполнится только в том случае, если значение переменной `count` больше, чем значение переменной `COUNT_LIMIT`. В противном случае оно игнорируется.

Реальные примеры программ позволяют лучше понять, как можно программно выразить процесс принятия решений. Предположим, что вы хотите написать программу, которая принимает с терминала целое число и затем отображает абсолютное значение. Это очень просто сделать, если сравнивать значение числа с нулем. Если число меньше нуля, то его преобразовывают в положительное число. Часть фразы “если число меньше нуля” свидетельствует о том, что в программе должно приниматься решение. Описать принятие решения в программе можно с помощью оператора `if`, как это показано в листинге 6.1.

Листинг 6.1. Расчет абсолютного значения целого числа

```
// В программе производится расчет абсолютного значения целого числа.
int main (void)
{
    int number;

    printf ("Type in your number: ");
    scanf ("%i", &number);
    if ( number < 0 )
        number = -number;
    printf ("The absolute value is %i\n", number);

    return 0;
}
```

Листинг 6.1. Вывод

```
Type in your number: -100
The absolute value is 100
```

Листинг 6.1. Вывод (Повторение)

```
Type in your number: 2000
The absolute value is 2000
```

Программа запускалась дважды с целью проверить ее работу при различных входных данных. Разумеется, обычно программу запускают достаточно много раз, чтобы убедиться в том, что она работает надежно и правильно. В нашем случае вводились различные возможные значения, на основании которых был сделан вывод, что программа функционирует правильно.

После того как отображается приглашение к вводу данных и введенное значение сохранено в переменной `number`, программа производит сравнение значения в переменной `number` с нулем. Если это значение меньше нуля, то выполняется следующее за оператором `if` утверждение, в котором это значение преобразуется в положительное значение. Если значение переменной `number` больше нуля, то утверждение, в котором производится преобразование в положительное значение, пропускается (нет необходимости его преобразовывать, т.к. оно является положительным). Затем абсолютное

значение переменной `number` отображается на экране и выполнение программы завершается.

Теперь рассмотрим программу из листинга 6.2, в которой также используется оператор `if`. Предположим, что вы получили несколько значений температуры, для которых вы хотите рассчитать среднее значение. Кроме того, необходимо составить список ошибочных значений.

В нашем случае предположим, что значения меньше чем 65 градусов (по Фаренгейту) являются ошибочными. Поэтому при составлении списка ошибочных значений необходимо будет принимать решение, является ли очередное значение ошибочным. При этом также будет использоваться оператор `if`.

Листинг 6.2. Расчет среднего значения и количества ошибочных данных

```
/* Расчет среднего значения и кол-ва ошибочных данных. */
#include <stdio.h>
int main (void)
{
    int numberOfGrades, i, grade;
    int gradeTotal = 0;
    int failureCount = 0;
    float average;

    printf ("How many grades will you be entering? ");
    scanf ("%i", &numberOfGrades);

    for ( i = 1; i <= numberOfGrades; ++i ) {
        printf ("Enter grade #i: ", i);
        scanf ("%i", &grade);
        gradeTotal = gradeTotal + grade;
        if ( grade < 65 )
            ++failureCount;
    }
    average = (float) gradeTotal / numberOfGrades;

    printf ("\nGrade average = %.2f\n", average);
    printf ("Number of failures = %i\n", failureCount);

    return 0;
}
```

Листинг 6.2. Вывод

```
How many grades will you be entering? 7
Enter grade #1: 93
Enter grade #2: 63
Enter grade #3: 87
Enter grade #4: 65
Enter grade #5: 62
Enter grade #6: 88
Enter grade #7: 76
Grade average = 76.29
Number of failures = 2
```

Переменная `gradeTotal`, которая используется для сохранения общей суммы введенных с терминала данных, инициализируется значением 0. Количество ошибочных данных сохраняется в переменной `failureCount`, которая также инициализируется значением 0. Переменная `average` объявлена как тип `float`, поскольку среднее значение последовательности целых чисел чаще всего не будет целым числом.

Сначала программа спрашивает пользователя о количестве вводимых значений и сохраняет его в переменной `numberOfGrades`. Это значение будет использоваться в цикле для подсчета среднего значения, в котором производится запрос на ввод каждого отдельно значения. Введенное значение сохраняется в переменной, для которой выбрано достаточно подходящее имя — `grade` (градус).

Значение, присвоенное переменной `grade`, затем добавляется в переменную `gradeTotal`, после чего производится проверка значения на достоверность. Если значение ошибочное, то значение переменной `failureCount` увеличивается на 1. Внешний цикл повторяется для ввода очередного значения из входных данных.

Когда все значения будут введены и просуммированы, в программе производится расчет среднего значения. Утверждение для расчета среднего значения может быть следующим.

```
average = gradeTotal / numberOfGrades;
```

Однако среднее значение, рассчитанное по такой формуле, обычно не является точным, так как теряется дробная часть. Это обусловлено тем, что если делимое и делитель являются целыми числами, то производится целочисленное деление и в результате получается целое число.

Создавшееся затруднение можно преодолеть двумя путями. Можно объявить переменную `numberOfGrades` или переменную `gradeTotal` как тип `float`. В этом случае будет производиться деление вещественных чисел и дробная часть не будет потеряна. Но такое решение нельзя назвать удачным, т.к. переменные `numberOfGrades` и `gradeTotal` используются для хранения целочисленных значений и объявление их в качестве переменных типа `float` не является корректным. Это не только вносит путаницу в программу, но и может привести к ошибкам.

Второй способ состоит в непосредственном приведении типа переменной к нужному типу в нужном месте. Для этого используется оператор преобразования типа (`float`), который и приводит тип переменной `gradeTotal` к типу `float` в процессе вычислений. Поскольку тип переменной `gradeTotal` приводится к типу `float` до того, как будет произведено деление, то оно будет выполняться как деление вещественных чисел. Это означает, что в результате деления дробная часть не будет отбрасываться.

После того как среднее значение будет рассчитано, оно отображается на терминале как вещественное число с двумя цифрами в дробной части. Для этого в утверждении `printf` в строке форматирования используется точка с последующим числом 2, которые ставятся непосредственно перед символом форматирования `f` (или `e`), а значит соответствующее значение должно отображаться с двумя дробными цифрами в дробной части. Точка с последующим числом называется *модификатором точности* и используется для задания формата выводимого вещественного числа.

Выполнение программы завершится после того, как будет отображено кол-во ошибочных значений.

Оператор if-else

Определяя число как четное или нечетное, вы, вероятнее всего, проверяете последнюю цифру этого числа. Если это цифра 0, 2, 4, 6 или 8, то вы делаете вывод, что это число четное. В противном случае число считается нечетным.

Но простейшим способом решения такой задачи с помощью компьютера будет не выделение последней цифры и ее анализ, а деление числа на 2. Если будет остаток от деления, то число нечетное, в противном случае число четное.

Вы уже знакомы с оператором деления по модулю (%), который используется для получения остатка при целочисленном делении. Именно этот принцип реализуется при определении четности числа. Производится деление по модулю 2 заданного числа и анализируется остаток. Если остаток равняется нулю, то число четное, в противном случае число нечетное.

В приведенном ниже листинге 6.3 написан код для определения четности числа, введенного пользователем, и отображения соответствующего сообщения на терминале.

Листинг 6.3. Определение четности числа

```
// Программа определяет, является ли введенное пользователем число
// четным.
#include <stdio.h>
int main (void)
{
    int number_to_test, remainder;

    printf ("Enter your number to be tested.: ");
    scanf ("%i", &number_to_test);

    remainder = number_to_test % 2;

    if ( remainder == 0 )
        printf ("The number is even.\n");
    if ( remainder != 0 )
        printf ("The number is odd.\n");

    return 0;
}
```

Листинг 6.3. Вывод

```
Enter your number to be tested: 2455
The number is odd.
```

Листинг 6.3. Вывод (Повторение)

```
Enter your number to be tested: 1210
The number is even.
```

После того как пользователь ввел число, производится деление по модулю 2 и анализ остатка. В первом утверждении `if` производится проверка остатка на равенство нулю. Если остаток равен нулю, то отображается сообщение "The number is even" (Число четное).

Во втором утверждении `if` производится проверка остатка на неравенство нулю. Если остаток не равен нулю, то отображается соответствующее сообщение, подтверждается что число нечетное. На самом деле, если первый оператор `if` будет соответствовать истине, то второе утверждение автоматически будет пропущено и наоборот.

Вспомните, ведь если число четное, то остаток от деления по модулю будет равен нулю, а другое значение он примет только в случае, если число нечетное.

При логических рассуждениях обычно используется слово "else" (то), поэтому конструкция с этим словом используется почти во всех языках программирования. В языке программирования C эта конструкция называется оператором if-else, ее формат представлен ниже.

```
if ( expression )
    program statement 1
else
    program statement 2
```

Оператор if-else действительно выражает наиболее общую форму утверждения if. Если результатом вычисления логического выражения expression будет истина (TRUE), то выполняется утверждение program statement 1, которое следует сразу за логическим выражением. В противном случае выполняется утверждение program statement 2. В любом случае, будет выполнено или утверждение program statement 1, или program statement 2, но не оба сразу.

Вы можете использовать оператор if-else в программе 6.3, заменяя два утверждения if одним оператором if-else. Используя эту программную конструкцию, вы сделаете программу более простой и читабельной, что видно из приведенного ниже листинга 6.4.

Листинг 6.4. Улучшенная программа определения четности числа

```
// Программа определяет, является ли введенное пользователем число
// четным. (Версия 2.)
#include <stdio.h>
int main ()
{
    int number_to_test, remainder;

    printf ("Enter your number to be tested: ");
    scanf ("%i", &number_to_test);

    remainder = number_to_test % 2;

    if ( remainder == 0 )
        printf ("The number is even.\n");
    else
        printf ("The number is odd.\n");

    return 0;
}
```

Листинг 6.4. Вывод

```
Enter your number to be tested: 1234
The number is even.
```

Листинг 6.4. Вывод (Повторение)

```
Enter your number to be tested: 6551
The number is odd.
```

Запомните, что два знака равенства подряд (==) являются оператором сравнения, а одиночный знак равенства является оператором присваивания. Вы потеряете впоследствии много времени и у вас возникнут большие проблемы в программе, если в логическом выражении `expression` утверждения `if` вы случайно вместо оператора сравнения поставите оператор присваивания.

Составные операторы отношения

В операторах `if`, которые вы изучали в этой главе, вы использовали только простые операторы отношений между двумя числами. В программе 6.1 вы сравнивали значение переменной `number` с нулем, а в программе 6.2 производилось сравнение значения переменной `grade` с числом 65. Но иногда возникает необходимость в более сложных сравнительных проверках. Предположим, например, что в программе 6.2 вы хотите подсчитать не кол-во ошибочных отсчетов, а кол-во значений, которые размещаются между числами 70 и 79. В этом случае вам придется проводить сравнение значения переменной `grade` не с одним значением, а с двумя — 70 и 79. То есть вы должны будете подсчитать все числа, которые попадают в заданный диапазон.

В языке программирования C предусмотрены специальные конструкции для проверки таких сложных зависимостей. Составные операторы отношений представляют собой один или несколько простых операторов отношений, объединенных с помощью логических операций `AND` (и) или `OR` (или). Эти операторы представлены двумя парами символов “&&” и “||” (две вертикальные черты) соответственно. Как пример можно привести следующий оператор

```
if ( grade >= 70 && grade <= 79 )
    ++grades_70_to_79;
```

где значение переменной `grades_70_to_79` инкрементируется только в том случае, если значение переменной `grade` больше или равно числу 70 и меньше или равно 79.

Аналогичным образом, оператор

```
if ( index < 0 || index > 99 )
    printf ("Error - index out of range\n");
```

приводит к выполнению утверждения `printf` только в том случае, если значение переменной `index` меньше чем 0 или больше чем 99.

Составные операторы можно использовать для формирования довольно сложных выражения ни языке программирования C. Язык C предоставляет программисту большую гибкость в формировании выражений. Часто наличие такой гибкости приводит к написанию довольно запутанных выражений, которые трудно воспринимать и отлаживать.

При написании сложных составных операторов старайтесь чаще использовать круглые скобки для получения более наглядного и лучше понятного выражения. Это позволит вам избежать многих ошибок. Также можно использовать пробелы с целью добиться большей выразительности при оформлении выражения. Дополнительные пространства вокруг операторов `&&` и `||` позволяют визуально выделить в выражении отдельные блоки, которые объединяются с помощью этих операторов.

Для демонстрации использования сложных составных операторов приведем работающую программу, которая позволяет рассчитать, является ли указанный год високосным. Високосными являются те года, которые без остатка делятся на 4, При этом они не должны без остатка делиться на 100, если они делятся без остатка на 400.

Подумайте, как бы вы могли реализовать проверку всех этих условий. Во-первых, вы должны подсчитать остатки от деления указанного года на 4, 100 и 400 и присвоить эти остатки специально созданным переменным, таким как `rem_4`, `rem_100` и `rem_400`. Затем необходимо провести проверку этих остатков на соответствие заданным критериям, после чего можно сделать вывод о том, является ли год високосным.

Если перефразировать предыдущее определение високосного года, то можно сказать, что високосным год будет только в том случае, если он без остатка делится на 4 и не делится без остатка на 100, или делится без остатка на 400. Подумайте, насколько это определение соответствует предыдущему определению и насколько его удобнее использовать для написания выражения. В таком виде его можно использовать для написания программного логического выражения, которое будет выглядеть следующим образом.

```
if ( (rem_4 == 0 && rem_100 != 0) || rem_400 == 0 )
    printf ("It's a leap year.\n");
```

Здесь необходимо учесть, что выражение.

```
rem_4 == 0 && rem_100 != 0
```

должно быть заключено в круглые скобки, поскольку в любом случае, его необходимо вычислять отдельно.

Если перед этими утверждениям добавить строки с объявлением переменных и ввода года с терминала, а после них добавить вывод необходимых сообщений, то получится программа, которая приведена в листинге 6.5.

Листинг 6.5. Определение високосного года

```
// Программа определяет, является ли указанный год високосным.
#include <stdio.h>
int main (void)
{
    int year, rem_4, rem_100, rem_400;

    printf ("Enter the year to be tested: ");
    scanf ("%i", &year);

    rem_4 = year % 4;
    rem_100 = year % 100;
    rem_400 = year % 400;

    if ( (rem_4 == 0 && rem_100 != 0) || rem_400 == 0 )
        printf ("It's a leap year.\n");
    else
        printf ("Nope, it's not a leap year.\n");

    return 0;
}
```

Листинг 6.5. Вывод

```
Enter the year to be tested: 1955
Nope, it's not a leap year.
```

Листинг 6.5. Вывод (Повторение)

```
Enter the year to be tested: 2000
It's a leap year.
```

Листинг 6.5. Вывод (Второе повторение)

```
Enter the year to be tested: 1800
Nope, it's not a leap year.
```

Итак, приведены три результата работы программы. В первом случае видно, что 1955-й год не является високосным, поскольку он не делится на 4 без остатка. Високосным является 2000 год поскольку он делится на 400 без остатка, и год 1800-й год не является високосным: хотя он и делится на 100 без остатка, но не делится без остатка на 400. Для полной проверки программы нужно еще подобрать год, который делится без остатка на 4, но не делится на 100. Это мы оставляем читателю в качестве упрощения.

Как уже отмечалось, язык программирования C обеспечивает большую гибкость при формировании выражений. Например, в предыдущей программе вы можете не рассчитывать отдельно значения переменных `rem_4`, `rem_100` и `rem_400`, а выполнить все непосредственно в логическом выражении для утверждения `if`.

```
if ( ( year % 4 == 0 && year % 100 != 0 ) || year % 400 == 0 )
```

Использование пробелов позволит выделить отдельные операции и сделает выражение более читабельным. Если вы будете игнорировать расстановку пробелов и удалите все дополнительные круглые скобки, то приведенное выше выражение будет выглядеть примерно следующим образом.

```
if (year%4==0&&year%100!=0||year%400==0)
```

Это выражение соответствует всем синтаксическим правилам и выполняется аналогично предыдущему выражению. Вам осталось только сравнить, какое из выражений легче для понимания.

Вложенные операторы if

Давайте вспомним общий формат оператора `if`, когда сначала производится вычисление логического выражения. Если результатом этого вычисления будет значение `TRUE`, то будет выполнено утверждение, непосредственно следующее за логическим выражением. При этом вполне допустимо, что это утверждение само является оператором `if`, как показано в примере ниже.

```
if ( gameIsOver == 0 )
    if ( playerToMove == YOU )
        printf ("Your Move\n");
```

При этом если значением переменной `gameIsOver` является 0, то будет выполнено следующее утверждение, которое является оператором `if`. В этом операторе `if` сравниваются значения переменных `playerToMove` и `YOU`. Если эти значения равны, то на терминале отображается сообщение "Your Move". Поэтому утверждение `printf` выполнится только в том случае, если значение переменной `gameIsOver` равно 0 и значение переменной `playerToMove` равно переменной `YOU`.

Эти два выражения можно записать и в виде одного составного выражения, как показано ниже.

```
if ( gameIsOver == 0 && playerToMove == YOU )
printf ("Your Move\n");
```

Более практичный пример использования вложенных операторов if получится после применения конструкции со словом else.

```
if ( gameIsOver == 0 )
    if ( playerToMove == YOU )
        printf ("Your Move\n");
    else
        printf ("My Move\n");
```

Выполнение этого утверждения в основном происходит так, как уже описывалось, за тем исключением, что если значение переменной gameIsOver равно 0 и значение переменной playerToMove не равно переменной YOU, то будет выполнено утверждение, которое следует за ключевым словом else. При этом на терминале будет отображено сообщение "My Move". Если значение переменной gameIsOver не равно 0, то все последующие операторы if, включая и связанное с ним ключевое слово else, пропускаются.

Обратите внимание, как ключевое слово else связывается с ключевым словом if. В последнем примере утверждение после слова else будет выполняться в том случае, если переменная playerToMove не равна переменной YOU, а не тогда, когда gameIsOver не равно 0. Общее правило таково: ключевое слово else всегда связано с последним ключевым словом if, которое еще не содержит слова else.

Вы можете дополнить предыдущий пример и добавить к внешнему оператору if ключевое слово else, переход на которое будет выполняться, если значение переменной gameIsOver не равно 0.

```
if ( gameIsOver == 0 )
    if ( playerToMove == YOU )
        printf ("Your Move\n");
    else
        printf ("My Move\n");
else
    printf ("The game is over\n");
```

Правильное использование отступов позволяет отразить логическую последовательность выполнения утверждений.

Разумеется, расстановка отступов нужна только для программиста, который таким образом выражает логику своих рассуждений. Компилятору эти отступы не нужны и он интерпретирует выражение в соответствии со своими внутренними правилами. Поэтому если программист расставил отступы, не совсем правильно понимая логические взаимосвязи, то полученный результат не будет соответствовать его ожиданиям. Например, если удалить первое ключевое слово else в предыдущем примере

```
if ( gameIsOver == 0 )
    if ( playerToMove == YOU )
        printf ("Your Move\n");
else
    printf ("The game is over\n");
```

то весь фрагмент будет интерпретирован не в соответствии с выполненным форматированием, а будет рассматриваться следующим образом

```

if ( gameOver == 0 )
    if ( playerToMove == YOU )
        printf ("Your Move\n");
    else
        printf ("The game is over\n");

```

поскольку ключевое слово `else` будет связано с последним ключевым словом `if`, не входящим в конструкцию `if-else`. Для того чтобы изменить логику рассуждений при наличии нескольких вложенных утверждений `if`, можно использовать фигурные скобки. Фигурные скобки как бы “закрывают” оператор `if`. Таким образом, в следующем фрагменте

```

if ( gameOver == 0 ) {
    if ( playerToMove == YOU )
        printf ("Your Move\n");
}
else
    printf ("The game is over\n");

```

будет достигнут требуемый результат и сообщение “The game is over” будет появляться тогда, когда значение переменной `gameIsOver` не равно 0.

Конструкция `else if`

Как вы уже могли убедиться, ключевое слово `else` удобно использовать, когда производится выбор из двух вариантов — число четное или нечетное, год високосный или не високосный. Однако программирование, конечно, гораздо сложнее упрощенного черно-белого мира. Рассмотрим программу, которая отображает -1, если введенное пользователем число меньше нуля, 0 — если число равно нулю, и 1 — если число больше нуля. (В действительности это будет реализация функции, которая называется знаковой функцией или функцией знака (*sign function*).) Очевидно, что в этом случае необходимо сделать три проверки — определить, будет ли введенное число меньше нуля, равно нулю или больше нуля. В этом случае оператор `if-else` не срабатывает. Конечно, всегда можно написать три последовательных утверждения `if`, но такое решение также не всегда будет работать, особенно если проверки являются взаимно исключающими.

Такую ситуацию можно описать на языке программирования C, если добавить оператор `if` после ключевого слова `else`, поскольку утверждение, которое следует за ключевым словом `else`, может быть любым разрешенным в языке C утверждением. Поэтому вполне логично, что это может быть и оператор `if`. В общем случае можно написать следующее

```

if ( expression 1 )
    program statement 1
else
    if ( expression 2 )
        program statement 2
    else
        program statement 3

```

и получить при этом расширение возможностей оператора `if-else` по обработке двух отдельных ситуаций для обработки трех ситуаций. Можно продолжать добавлять утверждения `if` после ключевых слов `else` и при этом получать эффективные способы для разрешения логических ситуаций с n вариантами.

Приведенная выше конструкция так часто используется при написании кода, что на нее обычно ссылаются как на конструкцию `else if` и обычно форматируют следующим образом.

```
if ( выражение 1 )
    утверждение 1
else if ( выражение 2 )
    утверждение 2
else
    утверждение 3
```

Такой метод форматирования способствует лучшему пониманию программы при чтении и при этом становится вполне ясно, что необходимо сделать выбор из трех ситуаций. В программе из листинга 6.6 проиллюстрировано использование конструкции `else. if` для реализации знаковой функции, описанной выше.

Листинг 6.6. Реализация знаковой функции

```
// Программа реализует знаковую функцию.
#include <stdio.h>
int main (void)
{
    int number, sign;

    printf ("Please type in a number: ");
    scanf ("%i", &number);

    if ( number < 0 )
        sign = -1;
    else if ( number == 0 )
        sign = 0;
    else          // Число положительное
        sign = 1;

    printf ("Sign = %i\n", sign);

    return 0;
}
```

Листинг 6.6. Вывод

```
Please type in a number: 1121
Sign = 1
```

Листинг 6.6. Вывод (Повторение)

```
Please type in a number: -158
Sign = -1
```

Листинг 6.6. Вывод (Второе повторение)

```
Please type in a number: 0
Sign = 0
```

Если введенное число меньше нуля, то переменная `sign` принимает значение `-1`; если число равно `0`, то значение переменной `sign` становится равным `0`, в остальных случаях введенное число должно быть больше нуля и переменная `sign` будет иметь значение `1`.

В программе из листинга 6.7 производится анализ символа, введенного с терминала, и его классификация как буквы (символы `a-z` или `A-Z`), цифры (`0-9`) или специального символа (все остальное). Для чтения одиночного символа с терминала используется утверждение `scanf` с символом форматирования `%c`.

Листинг 6.7. Классификация введенного с терминала символа

```
// В программе производится классификация введенного с терминала символа
#include <stdio.h>
int main (void)
{
    char c;

    printf ("Enter a single character:\n");
    scanf ("%c", &c);

    if ( (c >= 'a' && c <= 'z') || (c >= 'A' && c <= 'Z') )
        printf ("It's an alphabetic character.\n");
    else if ( c >= '0' && c <= '9' )
        printf ("It's a digit.\n");
    else
        printf ("It's a special character.\n");

    return 0;
}
```

Листинг 6.7. Вывод

```
Enter a single character:
&
It's a special character.
```

Листинг 6.7. Вывод (Повторение)

```
Enter a single character:
8
It's a digit.
```

Листинг 6.7. Вывод (Второе повторение)

```
Enter a single character:
В
It's an alphabetic character.
```

При первой проверке, которая производится сразу после считывания символа, определяется, является ли данный символ буквой. Это делается с помощью анализа диапазонов, в который должен входить данный символ. Данные диапазоны различны для строчных и заглавных букв. Сначала проверяется диапазон для строчных букв, т.е. вычисляется выражение

```
( c >= 'a' && c <= 'z' )
```

которое принимает значение TRUE, если значение переменной *c* находится в диапазоне символов от 'a' до 'z'. Это говорит о том, что значением переменной *c* является строчная буква. Затем вычисляется следующее выражение

```
( c >= 'A' && c <= 'Z' )
```

которое принимает значение TRUE в том случае, если значение переменной *c* находится среди символов от 'A' до 'Z'. Такие проверки возможны на всех компьютерах, которые используют для хранения символов формат ASCII.

Необходимо отметить, что в данном случае лучше всего использовать стандартные библиотечные функции `islower` и `isupper` и не вникать в подробности внутреннего представления символов. Данный пример приведен только в иллюстративных целях.

Если значением переменной *c* будет буква, то первая проверка пройдет успешно и на экране отобразится сообщение "It's an alphabetic character". Если результатом проверки будет значение FALSE, то начнет выполняться следующая конструкция `else if`. При этом будет производиться проверка относительно того, является ли значение переменной *c* цифрой, а также, попадает ли значение переменной *c* в диапазон символов от '0' до '9', а не цифр от 0 до 9. Это действительно необходимо, поскольку с терминала считываются коды для цифр, а не сами цифры. Для обозначения цифр в кодировке ASCII (о чем уже упоминалось выше) используются специальные коды, где цифре 0 присвоен код 48, цифре 1 — код 49 и т.д.

Если значением переменной *c* является цифровой символ, то сообщение "It's a digit" отобразится на экране. В противном случае значением переменной *c* не является ни буква, ни цифровой символ и на терминале отобразится сообщение "It's a special character". На этом выполнение программы завершается.

Обратите внимание на то, что если вы вводите с помощью утверждения `scanf` даже одиночный символ, то вы все равно должны нажать клавишу <Enter>, чтобы сообщить программе об окончании ввода. В любом случае, при вводе данных с терминала программа не может сама принять решение об окончании ввода, всегда необходимо нажимать клавишу <Enter>.

В следующем примере будет представлена программа, в которой пользователь может ввести простое выражение в виде

число оператор число

и получить результат. Программа рассчитывает выражение и отображает полученное число на экране терминала с точностью до двух десятичных цифр.

При этом вводимый оператор должен быть обычным оператором сложения, вычитания, умножения и деления. В программе из листинга 6.8 используется оператор `if` с несколькими конструкциями `else if` для определения того, какой оператор был введен пользователем.

Листинг 6.8. Расчет простого выражения

```
/* Программа расчета простого выражения */
#include <stdio.h>
int main (void)
{
    float value1, value2;
    char operator;

    printf ("Type in your expression.\n");
```

```

scanf ("%f %c %f", &value1, &operator, &value2);

if ( operator == '+' )
    printf ("%f\n", value1 + value2);
else if ( operator == '-' )
    printf ("%f\n", value1 - value2);
else if ( operator == '*' )
    printf ("%f\n", value1 * value2);
else if ( operator == '/' )
    printf ("%f\n", value1 / value2);

return 0;
}

```

Листинг 6.8. Вывод

```

Type in your expression.
123.5 + 59.3
182.80

```

Листинг 6.8. Вывод (Повторение)

```

Type in your expression.
198.7 / 26
7.64

```

Листинг 6.8. Вывод (Второе повторение)

```

Type in your expression.
89.3 * 2.5
223.25

```

В утверждении `scanf` указано, что должны быть считаны три значения в переменные `value1`, `operator` и `value2`. Для считывания вещественных переменных используется формат ввода `%f` (аналогичное форматирование используется при выводе вещественных значений). Такой формат используется для считывания значения в переменную `value1`, которая является первым операндом выражения.

Вторым операндом выражения следует считать оператор. Поскольку оператор является одним из символов `'+'`, `'-'`, `'*'` или `'/'` и не является числом, то должно производиться считывание символа в переменную `operator`. Для считывания символа используется символ форматирования `%c`. Пробелы внутри строки форматирования говорят о том, что произвольное число пробелов может встретиться при разделении вводимых данных. Это позволит разделить пробелами операнды и оператор при вводе данных с терминала. Если написать строку форматирования следующим образом: `"%f%c%f"`, то нельзя будет размещать пробелы между вводимыми данными, т.е. нельзя будет поставить пробел после первого операнда и после оператора, поскольку когда в утверждении `scanf` необходимо будет считать значение в соответствии с форматом `%c`. В результате будет просто считано очередное значение, даже если это будет пробел, который также имеет свой код. Но при этом необходимо отметить, что при выполнении утверждения `scanf` всегда игнорируются пробелы перед считыванием целых или вещественных чисел. Поэтому строка форматирования `"%f %c%f"` в нашем случае тоже будет работать правильно.

После того как второй операнд будет считан и сохранен в переменной `value2`, программа приступает к проверке значения переменной `operator` на соответствие одному из четырех значений. Когда обнаруживается совпадение, то выполняется соответствующее утверждение `printf` и на экране терминала отображается результат расчета. После этого выполнение программы завершается. Хотя это и не совсем так. В программе нет проверки на предмет того, правильно ли выполнен ввод входных данных и, например, что произойдет, если пользователь ошибочно введет символ "?" как символ оператора. Программа просто пропустит оператор `if`, и на экране не появится никакого сообщения, даже сообщения о том, что пользователь ввел неправильное выражение, которое невозможно рассчитать.

Или, например, пользователь вводит оператор деления со вторым операндом, равным нулю. Как вы уже знаете, деление на ноль в языке программирования C приведет к непредсказуемым результатам. В программе обязательно должна производиться проверка на предмет возникновения такой ситуации.

Поэтому всегда старайтесь предусмотреть все ситуации, чреватые непредсказуемыми результатами, и выполняйте необходимые действия для того, чтобы избежать таких ситуаций. Всегда старайтесь проверять программы со всеми возможными входными данными, что поможет устранить потенциальные ошибки. Но для устранения ошибок необходимо делать не только это. При написании программы необходимо придерживаться гипотетического правила: "Что может случиться, если ..." и при этом вставлять дополнительные строки для корректной обработки ситуации.

Ниже приведен листинг 6.8А с модифицированной программой из листинга 6.8, в котором предусмотрены ситуации деления на ноль и неправильного ввода символа оператора.

Листинг 6.8А. Улучшенная программа расчета простого выражения

```
/* Программа расчета простого выражения.
value operator value */
#include <stdio.h>
int main (void)
{
    float value1, value2;
    char operator;

    printf ("Type in your expression.\n");
    scanf ("%f %c %f", &value1, &operator, &value2);

    if ( operator == '+' )
        printf (".2f\n", value1 + value2);
    else if ( operator == '-' )
        printf (".2f\n", value1 - value2);
    else if ( operator == '*' )
        printf (".2f\n", value1 * value2);
    else if ( operator == '/' )
        if ( value2 == 0 )
            printf ("Division by zero.\n");
        else
            printf (".2f\n", value1 / value2);
    else
        printf ("Unknown operator.\n");

    return 0;
}
```

Листинг 6.8А. Вывод

```
Type in your expression.  
123.5 + 59.3  
182.80
```

Листинг 6.8А. Вывод (Повторение)

```
Type in your expression.  
198.7 / 0  
Division by zero.
```

Листинг 6.8А. Вывод (Второе повторение)

```
Type in your expression.  
125 $ 28  
Unknown operator.
```

Когда вводится оператор деления, т.е. обратная косая черта, то производится дополнительная проверка значения второго оператора `value2`, которое не должно быть равно нулю. Если оно равно нулю, то на терминале отображается соответствующее сообщение. В противном случае выполняется операция деления и на экране терминала отображается результат. Внимательно рассмотрите вложенные утверждения `if` и связанные с ними ключевые слова `else`. Обращение к ключевому слову `else` в конце программы производится во всех ошибочных ситуациях. Поэтому когда значение переменной `operator` не совпадает ни с одним из разрешенных значений, то происходит переход именно на это ключевое слово и выполняется вывод сообщения “Unknown operator”.

Оператор switch

Последовательность операторов `if-else`, которую вы вводили в последней программе, когда значения переменных последовательно сравниваются с различными значениями, настолько часто используется в различных программах, что для такой конструкции в языке программирования C разработан специальный оператор. Такой оператор называется `switch` и его формат представлен следующим образом.

```
switch ( expression )  
{  
    case value1:  
        program statement  
        program statement  
        ...  
        break;  
    case value2:  
        program statement  
        program statement  
        ...  
        break;  
    ...  
}
```

```

case valuen:
    program statement
    program statement
    ...
    break;
default:
    program statement
    program statement
    ...
    break;
}

```

Выражение `expression`, заключенное в круглые скобки, последовательно сравнивается со значениями `value1`, `value2`, ..., `valuen`, которые должны быть простыми константами или константными выражениями. В том случае, когда одно из этих значений равно значению выражения `expression`, выполняются те утверждения, которые следуют за данным значением. Обратите внимание, что таких утверждений может быть сколь угодно много и они не заключаются в фигурные скобки.

Утверждение `break` сигнализирует об окончании выполнения утверждений и приводит к выходу из оператора `switch`. Не забывайте ставить утверждение `break` в конце каждого варианта выбора. Если этого не сделать, то выполнение последовательности утверждений перейдет в следующий вариант выбора и будет выполняться до тех пор, пока не встретится утверждение `break`.

Специальный дополнительный вариант `default` будет выполнен в том случае, когда не будет найдено ни одного совпадения. Это эквивалентно переходу на ключевое слово `else` в предыдущем примере, когда введено неправильное значение оператора. Можно реализовать оператор `switch` с помощью утверждений `if` и получить эквивалентную конструкцию, как показано ниже.

```

if ( expression == value1 )
{
    program statement
    program statement
    ...
}
else if ( expression == value2 )
{
    program statement
    program statement
    ...
}
...
else if ( expression == valuen )
{
    program statement
    program statement
    ...
}
else
{
    program statement
}

```

```

    program statement
    ...
}

```

Зная это, можно перевести последовательность утверждений `if` из листинга 6.8А в аналогичный оператор `switch`, что и сделано в программе из листинга 6.9.

Листинг 6.9. Улучшенная программа расчета простого выражения, вариант 2

```

/* Программа расчета простого выражения.
value operator value */
#include <stdio.h>
int main (void)
{
    float value1, value2;
    char operator;

    printf ("Type in your expression.\n");
    scanf ("%f %c %f", &value1, &operator, &value2);
    switch (operator)
    {
        case '+':
            printf ("%f\n", value1 + value2);
            break;
        case '-':
            printf ("%f\n", value1 - value2);
            break;
        case '*':
            printf ("%f\n", value1 * value2);
            break;
        case '/':
            if ( value2 == 0 )
                printf ("Division by zero.\n");
            else
                printf ("%f\n", value1 / value2);
            break;
        default:
            printf ("Unknown operator.\n");
            break;
    }

    return 0;
}

```

Листинг 6.9. Вывод

```

Type in your expression.
178.99 - 326.8
-147.81

```

После того как выражение будет считано, значение переменной `operator` будет последовательно сравниваться со значениями, указанными для каждого случая. Если совпадение найдено, то начинают выполняться утверждения, относящиеся к данному случаю. При достижении утверждения `break` происходит выход из утверждения

switch и выполнение программы завершается. Если совпадений не обнаружено, то происходит переход к метке default и отображается сообщение "Unknown operator".

На самом деле можно не ставить утверждения break в конце ряда утверждений при выборе варианта default, т.к. при этом все равно будет сделан выход из утверждения switch. Тем не менее это должно стать хорошей привычкой — включать утверждение break в конце каждого варианта выбора.

Когда вы пишете оператор switch, имейте ввиду, что значения для различных ситуаций не должны совпадать. Однако вы можете связать несколько значений с одной ситуацией или с одной последовательностью утверждений. Это можно сделать простым перечислением значений с ключевым словом case для каждого значения, заканчивая это перечисление двоеточием перед группой утверждений, которые должны быть выполнены для данных значений. Например, в операторе switch, приведенном ниже, утверждение printf, в котором производится перемножение значений переменных value1 и value2, выполняется при равенстве значения переменной operator звездочке (*) или букве x в нижнем регистре.

```
switch (operator)
{
    ...
    case '*':
    case 'x':
        printf("%.2f\n", value1 * value2);
        break;
    ...
}
```

Булевы переменные

Большинство программистов обычно сталкиваются с задачей написания программы генерации таблицы простых чисел. Напомню, что положительное целое число p будет считаться простым числом, если оно не делится без остатка ни на какое другое число, кроме единицы и самого себя.

Первым простым числом будет число 2. Следующим простым числом будет число 3, поскольку оно не делится ни на что, кроме единицы и тройки. Но число 4 уже не будет простым числом, поскольку оно делится на 2.

Для получения таблицы простых чисел можно использовать различные алгоритмы. Например, если перед вами стоит задача получить все простые числа до числа 50, то наиболее легким способом будет выбор прямого алгоритма, когда каждое очередное число p проверяется делением на все числа от 2 до $p-1$. Если при делении на такие числа возможен хоть один вариант деления без остатка, то число не является простым, в противном случае, число простое. В программе из листинга 6.10 приведен именно такой алгоритм составления таблицы простых чисел.

Листинг 6.10. Составление таблицы простых чисел

```
// Программа генерирует последовательность простых чисел.
#include <stdio.h>
int main (void)
{
    int p, d;
    _Bool isPrime;
```

```

for ( p = 2; p <= 50; ++p ) {
    isPrime = 1;
    for ( d = 2; d < p; ++d )
        if ( p % d == 0 )
            isPrime = 0;
    if ( isPrime != 0 )
        printf ("%i ", p);
}
printf ("\n");

return 0;
}

```

Листинг 6.10. Вывод

2 3 5 7 11 13 17 19 23 29 31 37 41 43 47

Приведен несколько замечаний относительно этой программы. Внешний цикл `for` выполняется в соответствии с переменной цикла от 2 до 50. Переменная цикла `p` является тем значением, которое проверяется на принадлежность к простым числам. В первом утверждении цикла переменной `isPrime` присваивается значение 1. Вам вскоре станет понятным назначение этой переменной.

Второй цикл последовательно делит значение переменной `p` на числа от 2 до `p-1`. Внутри цикла производится проверка на предмет того, равен ли остаток от деления нулю или нет. Если равен, то это число не может быть простым числом, поскольку оно должно делиться только на 1 и на само себя. Свидетельством того, что очередное значение переменной `p` не является простым числом, будет присваивание переменной `isPrime` значения 0.

Когда внутренний цикл закончит выполнение, то проверяется значение переменной `isPrime`. Если ее значение не равно нулю, то это означает, что не было найдено чисел, деление на которые значения переменной `p` производилось без остатка, поэтому данное значение должно быть простым числом, и именно это значение отображается на экране.

Вы, наверное, обратили внимание, что переменная `isPrime` принимает только два значения: 0 и 1, и ничего больше. Именно поэтому она и была объявлена как переменная типа `_Bool`. Ее значение, равное 1, говорит о наличии простого числа. Но как только будет получено деление без остатка, значение переменной `isPrime` станет равным нулю, а значит, очередное значение переменной `p` не является простым числом.

Очень часто переменные, которые используются подобным образом, называются флагами. Флаг обычно принимает одно из двух различных значений. Поэтому значение флага проверяется в программе на наличие двух положений: “установлен” (TRUE) или “сброшен” (FALSE), и в зависимости от положения флага выполняются различные действия.

В языке программирования C положения флага TRUE (истина) и FALSE (ложь) отображаются значениями 1 и 0 соответственно. Поэтому когда в программе из листинга 6.10 значение переменной `isPrime` становится равным 1, то это воспринимается как истина (TRUE) и число считается простым. Если в процессе выполнения внутреннего цикла деления без остатка не найдено, то значение переменной `isPrime` говорит о том, что предположение о простом числе ложно (FALSE).

Это просто совпадение, что значение 1 обычно используется для представления TRUE, или “установлен”, а значение 0 представляет FALSE, или “сброшен”. Это

представление ассоциируется с битом — когда бит установлен, то его значение равно 1, а когда сброшен, его значение равно 0. Но в языке программирования С несколько шире трактуются значения для логических переменных. Рассмотрим подробнее концепцию логических значений TRUE и FALSE.

Вспомним начало этой главы, когда мы рассматривали выражения условия для утверждения `if`. Если условие удовлетворялось, то выполнялось утверждение, непосредственно следующее за выражением условия.

Но что конкретно означает слово “удовлетворяет”. В языке программирования С слово “удовлетворяет” означает, что значение не равно 0. Поэтому выполнение утверждения

```
if ( 100 )
printf ("This will always be printed.\n");
```

закончится тем, что будет выполнено утверждение `printf`, поскольку значение выражения условия в утверждении `if` не равно 0 (в нашем случае это число 100), и поэтому оно считается удовлетворительным.

В каждой программе этой главы используются утверждения “не нуль означает удовлетворительно” и “нуль означает неудовлетворительно”. Применительно к языку программирования С, это свидетельствует о том, что результатом вычисления выражения условия может быть значение 1, что означает удовлетворительно, и значение 0, что означает неудовлетворительно. Поэтому при вычислении выражения

```
if ( number < 0 )
number = -number;
```

выполняются следующие действия.

1. Вычисляется выражение условия `number < 0`. Если условие удовлетворяется, что произойдет, когда при значении переменной `number` станет меньше нуля, то результатом вычисления будет значение 1, в противном случае это будет значение 0.
2. В утверждении `if` производится проверка результата вычисления. Если результат не равен нулю, то выполняется утверждение, следующее непосредственно за выражением условия, в противном случае это выражение пропускается.

Предыдущие рассуждения также применимы и к утверждениям `for`, `while` и `do`. Расчет сложного выражения условия наподобие.

```
while ( char != 'e' && count != 80 )
```

также производится в соответствии с предыдущими рассуждениями. Если оба условия удовлетворяются, результатом вычисления будет 1, но если хотя бы одно условие не выполняется, то результатом будет 0. Результат вычисления всегда проверяется. Если это значение 0, то происходит выход из цикла `while`. В противном случае начинается очередной цикл вычислений.

Вернемся к программе из листинга 6.10 и понятию флага. В языке программирования С можно проверять состояние флага следующим образом.

```
if ( isPrime )
```

и это полностью эквивалентно следующему выражению.

```
if ( isPrime != 0 )
```

Для того чтобы убедиться, что состояние флага равно FALSE, можно использовать оператор логического отрицания (!). В выражении

```
if ( ! isPrime )
```

оператор логического отрицания используется для проверки того, что значение переменной isPrime равно FALSE. В общем, все выражения наподобие

```
! expression
```

отрицают логическое значение выражения expression. Поэтому если значение выражения expression равно нулю, то оператор логического отрицания превратит его в 1. А если значение выражения expression не равно нулю, то оператор логического отрицания превратит его в 0.

С помощью оператора логического отрицания можно легко изменить значение флага.

```
myMove = ! myMove;
```

Следовательно, этот оператор имеет такой же приоритет, как и унарный минус. Это означает, что он имеет более высокий приоритет, чем все бинарные арифметические операторы и все операторы отношения. Поэтому проверить, что значение переменной x не меньше значения переменной y, можно следующим образом.

```
! ( x < y )
```

Круглые скобки необходимы для того, чтобы выражение было корректным. Конечно, можно написать эквивалентное выражение следующим образом.

```
x >= y
```

В главе 4 “Переменные, типы данных и арифметические выражения”, вы узнали о некоторых специальных значениях, которые используются в языке программирования C и которые можно использовать при работе с булевыми значениями. Это тип bool и значения TRUE и FALSE. Для того чтобы ими можно было пользоваться, необходимо подключить к вашей программе заголовочный файл stdbool.h. Программа из листинга 6.10A полностью соответствует программе из листинга 6.10, но переписана с использованием этих типов и значений.

Листинг 6.10A. Улучшенная программа составления таблицы простых чисел

```
// Программа генерирует последовательность простых чисел.
#include <stdio.h>
#include <stdbool.h>
int main (void)
{
    int p, d;
    bool isPrime;

    for ( p = 2; p <= 50; ++p ) {
        isPrime = true;
        for ( d = 2; d < p; ++d )
            if ( p % d == 0 )
                isPrime = false;
        if ( isPrime != false )
            printf ("%i ", p);
    }
}
```

```
printf ("\n");

return 0;
}
```

Листинг 6.10А. Вывод

2 3 5 7 11 13 17 19 23 29 31 37 41 43 47

Вполне очевидно, что, подключив к программе заголовочный файл `stdbool.h`, вы можете объявлять переменные типа `bool` вместо типа `_Bool`. Это лишь “косметические” улучшения, поскольку так удобнее писать и понимать программу, и это больше согласуется со стилем написания других типов данных языка C, таких как `int`, `float` и `char`.

Оператор условия

Возможно, самым необычным оператором в языке программирования C можно назвать оператор условия. В отличие от других операторов языка C, которые могут быть унарными или бинарными, оператор условия является *тернарным* оператором. Это означает, что это у него может быть три операнда. Два символа, которые используются при написании данного оператора, — это знак вопроса (?) и двоеточие (:).

Первый операнд размещается перед знаком вопроса, второй размещается между знаком вопроса и двоеточием, а третий ставится после двоеточия.

Общий формат оператора условия представлен ниже.

```
условие ? выражение_1 : выражение_2
```

Здесь условие является выражением, обычно выражением условия, которое рассчитывается в начале выполнения оператора условия. Если в результате вычисления условия будет получено значение `TRUE` (не ноль), то выполняется выражение_1, и результатом выполнения оператора условия будет значение, полученное при вычислении этого выражения. Если в результате вычисления условия будет получено значение `FALSE` (ноль), то выполняется выражение_2, и результатом выполнения оператора условия будет значение, полученное при вычислении выражения выражение_2.

Оператор условия наиболее часто используется для присвоения переменной одного из двух значений в зависимости от некоторого условия. Например, предположим, что вы имеете целочисленные переменные `x` и `s`. Если вы хотите присвоить значение `-1` переменной `s` тогда, когда значение переменной `x` меньше нуля, и присвоить переменной `s` значение `x2` в противном случае, то можно написать следующее утверждение.

```
s = ( x < 0 ) ? -1 : x * x;
```

При выполнении этого утверждения сначала проверяется условие `x<0`. В круглые скобки заключается выражение условия для того, чтобы улучшить читабельность утверждения. Хотя обычно круглые скобки не требуются, т.к. приоритет оператора условия невысокий и все операторы старше его, за исключением операторов присваивания и запятой. Если значение `x` меньше нуля, то выполняется выражение, непосредственно следующее за знаком вопроса. В нашем случае это просто целочисленное

значение -1 , которое и присваивается переменной s . Если значение переменной x не меньше нуля, выполняется выражение, непосредственно следующее за двоеточием, и переменной s присваивается результат его вычисления. Этим выражением будет $x * x$ или x^2 .

Рассмотрим еще один пример использования оператора условия, когда переменной $maxValue$ должно быть присвоено большее из двух значений a или b , т.е. должно быть выполнено следующее утверждение.

```
maxValue = ( a > b ) ? a : b;
```

Если выражение, которое поставлено после двоеточия (аналог конструкции “else”), состоит из другого оператора условия, то можно получить эффект конструкции “else if”. Например, знаковая функция, которая была реализована в программе из листинга 6.6, может быть написана в программе одной строкой с помощью двух операторов условия следующим образом.

```
sign = ( number < 0 ) ? -1 : (( number == 0 ) ? 0 : 1);
```

Рассмотрим последовательность выполнения. Если значение переменной $number$ меньше нуля, то переменной $sign$ присваивается значение -1 и выполнение утверждения заканчивается. В противном случае, когда значение переменной $number$ больше или равно нулю, будет выполняться второй оператор, условия и если значение переменной $number$ равно нулю, то переменной $sign$ присваивается значение 0 , в противном случае переменной $sign$ присваивается значение 1 . В данном утверждении второй оператор условия, т.к. не нужно заключать в круглые скобки оператор условия выполняется справа налево, а конструкция с несколькими операторами условия, такая как представленная ниже

```
e1 ? e2 : e3 ? e4 : e5
```

группируется справа налево и выполняется следующим образом.

```
e1 ? e2 : ( e3 ? e4 : e5 )
```

Оператор условия необязательно должен использоваться только с правой стороны оператора присваивания, он может использоваться везде, где можно написать выражение. Это означает, что можно отобразить знак переменной $number$ без присваивания результата отдельной переменной, а непосредственно в утверждении `printf`, как показано ниже.

```
printf ("Sign = %i\n", ( number < 0 ) ? -1 : ( number == 0 ) ? 0 : 1);
```

Оператор условия удобно использовать в языке программирования C при написании препроцессорных *макроопределений*. С подробностями можно ознакомиться в главе 13, “Препроцессор”.

На этом завершается обсуждение вопросов относительно принятия решений. В главе 7, “Работа с массивами”, вы познакомитесь с более сложными типами данных. Массив является мощным средством, которое вы обязательно будете использовать при написании своих программ на языке программирования C.

Упражнения

1. Наберите и запустите 10 программ, представленные в этой главе. Сравните выводы, сделанные вашими программами, с выводами, представленными после листингов программ. Поэкспериментируйте с каждой программой, вводя различные значения.
2. Напишите программу, запрашивающую пользователя ввести два целочисленных значений. Определите, делится ли первое число на второе без остатка, и выведите соответствующее сообщение на экран.
3. Напишите программу, которая принимает от пользователя два целочисленных значения. Отобразите результат деления первого значения на второе с точностью до трех десятичных знаков. Помните о том, что может произойти деление на ноль.
4. Напишите программу, которая работает как простейший калькулятор. Программой должны распознаваться следующие операторы.

"+", "-", "*", "/", "S", "E".

Оператор S используется для сохранения введенного числа в аккумуляторе. Оператор E используется для выхода из программы. Арифметические операции выполняются с содержимым аккумулятора и числом, введенным пользователем. Ниже приведены примеры того, касающиеся программы работы.

Начало вычислений

```
10 S      { Сохранить в аккумуляторе число 10. }
= 10.000000 { Содержимое аккумулятора. }
2 /       { разделить на 2. }
= 5.000000  { Содержимое аккумулятора. }
55 -      { Вычесть 55 }
-50.000000
100.25 S   { Сохранить в аккумуляторе число 100.25. }
= 100.250000
4 *        { Умножить на 4. }
= 401.000000}
0 E        { Конец программы. }
= 401.000000
```

Конец вычислений.

Необходимо учесть деление на ноль и распознавание неправильно введенных операторов.

5. В главе 5 была разработана программа реверсирования цифр числа и вывод результата на терминал. Однако, эта программа не работает корректно, если ввести отрицательное число. Проанализируйте, что происходит в этом случае и измените программу таким образом, чтобы отрицательные числа обрабатывались корректно. Например, если ввести число -8645, то на экране должна отобразиться последовательность 5468-.
6. Напишите программу, которая принимает число, введенное пользователем с экрана, и выделяет и отображает значение каждой цифры числа на английском

языке. Например, если пользователь наберет число 982, программа должна напечатать:

```
nine   three   two
```

Не забывайте сделать вывод слова “zero”, если пользователь введет только число 0. Заметим, что это довольно сложное упражнение.

7. Программа из листинга 6.10 имеет несколько неточностей, снижающих эффективность работы программы. Во-первых, можно не проверять четные числа. Очевидно, что все четные числа, большие 2, не могут быть простыми числами, поэтому можно не производить проверку всех четных чисел, которые априори не являются простыми. Внутренний цикл также не является эффективным, поскольку значения переменной p всегда делятся на все значения переменной d от 2 до $p-1$. Это можно исправить, если производить проверку значения переменной `isPrime` в выражении условия для цикла. Модифицируйте эту программу с целью исключить эту неточность. Запустите программу и проверьте все операции. (В главе 7 вы узнаете о еще более эффективном способе генерации простых чисел.)

Массивы

В языке программирования C заложены средства для задания последовательностей упорядоченных данных. Такие последовательности называются *массивами*. В данной главе описывается, как можно объявить массив и как им управлять. В последующих главах вы узнаете еще больше о массивах и познакомитесь с тем, как массивы работают с функциями, структурами, символьными строками и указателями.

Предположим, что вы имеете множество отсчетов *grades*, которые вы вводите в компьютер и с которыми вы будете выполнять некоторые операции, такие как ранжирование, подсчет среднего значения или определение медианы. В программе из листинга 6.2 вы уже рассчитывали среднее значение множества данных, просто добавляя каждый вновь введенный отсчет к общей сумме. Однако если вы захотите упорядочить множество *grades* в возрастающем порядке, то вы не сможете этого сделать, пока не введете все значения полностью. Поэтому вы должны ввести каждое значение и сохранить его в отдельной переменной, например, с помощью следующего ряда утверждений.

```
printf ("Enter grade 1\n");
scanf ("%i", &grade1);
printf ("Enter grade 2\n");
scanf ("%i", &grade2);
. . .
```

Только после того, как все значения будут введены, вы можете выполнить ранжирование. Это можно сделать с помощью повторяющихся операторов *if*, сравнивая пары значений и определяя наименьшее значение, затем сравнивая наименьшее с последующим значением и т.д., пока не будет найдено самое наименьшее значение. Но если вы попытаетесь написать программу для упорядочивания множества значений, то скоро убедитесь, что используя только последовательности операторов *if*, можно выполнить ранжирование только для небольшого числа значений, в противном случае программа получается большой и сложной. Но не все так плохо, и именно благодаря массивам.

Объявление массива

Вы можете использовать переменную с именем `grades` для хранения не только одного значения, а всего множества значений отсчетов. На каждый элемент множества можно сослаться с помощью индексной переменной или *индекса* `index`. В математике для выделения отдельного элемента используется подстрочный индекс, и запись x_i ссылается на порядковый элемент с номером i . В языке программирования C это записывается следующим образом.

```
x[i]
```

Выражение

```
grades[5]
```

ссылается на пятый элемент массива с именем `grades`. Индексация элементов начинается с нулевого значения, поэтому выражение

```
grades[0]
```

реально ссылается на первый элемент массива. При этом проще рассуждать в терминах ссылок на элемент с индексом нуль, чем ссылаться на первый элемент. Каждая переменная типа массив может использоваться везде, где используется обычная переменная. Например, вы можете использовать значение переменной типа массив для присвоения значений другой переменной с помощью следующего утверждения.

```
g = grades[50];
```

В этом случае значение элемента `grades[50]` присваивается переменной `g`. В более общем случае, если переменная `i` объявлена как целочисленная переменная, то утверждение

```
g = grades[i];
```

присвоит значение элемента с номером `i` массива `grades` переменной `g`. Поэтому если значение переменной `i` равно 7, то в предыдущем утверждении значение `grades[7]` будет присвоено переменной `g`.

Значение элементу массива можно присвоить с помощью оператора присваивания, просто указав с левой стороны оператора необходимое значение. В утверждении

```
grades[100] = 95;
```

значение 95 присваивается и сохраняется в сотом элементе массива `grades`. В утверждении

```
grades[i] = g;
```

происходит сохранение значения переменной `g` в элементе массива `grades[i]`.

Способность представлять набор отдельных элементов с помощью одного массива позволяет создавать лаконичные и эффективные программы. Например, вы можете просмотреть последовательность элементов в массиве, только изменяя значение переменной, которая используется как индексная переменная. Поэтому в цикле `for`

```
for ( i = 0; i < 100; ++i )  
    sum += grades[i];
```

будут суммированы первые 100 элементов массива `grades` (элементы от 0 до 99) и значение суммы будет сохранено в переменной `sum`. По окончании цикла в переменной

sum будет находиться значение суммы первых 100 элементов массива grades при условии, что до начала цикла эта переменная имела значение 0.

При работе с массивами помните, что первый элемент массива имеет индекс ноль, а индекс последнего элемента массива имеет значение, равное числу элементов минус один. В дополнение к целочисленным константам, целочисленные выражения также могут использоваться в квадратных скобках для ссылки на отдельный элемент массива. Поэтому если переменные low и high являются целочисленными переменными, то утверждение

```
next_value = sorted_data[(low + high) / 2];
```

присваивает значение элемента массива с индексом, полученным в результате расчета выражения $(low+high)/2$, переменной next_value. Если значение переменной low равно 1, а значение high равно 9, то переменной next_value будет присвоено значение элемента массива sorted_data[5]. Причем если значение переменной low равно 1, а значение high равно 10, то переменной next_value будет также присвоено значение элемента массива sorted_data[5], т.к. целочисленное деление 11 на 2 дает значение 5.

Аналогично переменным, массивы могут объявляться перед тем местом программы, где они будут использоваться. При объявлении массива необходимо указать тип элементов, которые будут составлять массив, — такой как int, float или char, и максимальное число элементов, которые будут сохраняться в массиве. Последняя информация необходима для компилятора, который должен знать, сколько памяти зарезервировать для массива.

Например, объявим массив следующим образом.

```
int grades[100];
```

это говорит о том, что в массиве grades будет содержаться 100 целочисленных элементов. Допустимая ссылка на элементы массива может быть выполнена с помощью значений индекса от 0 до 99. Поэтому будьте осторожны при задании значений индексной переменной, компилятор языка программирования C не производит проверки диапазона значений индексной переменной. И если к объявленному выше массиву grades вы будете обращаться в помощью индекса 150, то компилятор пропустит это и в программе не возникнет аварийной ситуации, но полученное значение будет абсолютно непредсказуемым и конечный результат расчета также будет неверным.

Для объявления массива с именем averages, который должен содержать 200 вещественных значений, используется следующее утверждение.

```
float averages[200];
```

При этом происходит выделение достаточного количества памяти для того, чтобы можно было вместить 200 вещественных чисел. Аналогично, объявление

```
int values[10];
```

зарезервирует достаточный объем памяти для массива с именем values чтобы могло разместиться 10 целых чисел. Размещение элементов массива в памяти представлено на рис. 7.1.

С элементами массива, объявленными как тип int, float или char, можно работать как с обычными переменными. Им можно присваивать значения, отображать их значения, суммировать, вычитать и т.д. Поэтому если в программе появятся утверждения, приведенные ниже, то массив будет содержать значения, которые показаны на рис. 7.2.

values [0]	
values [1]	
values [2]	
values [3]	
values [4]	
values [5]	
values [6]	
values [7]	
values [8]	
values [9]	

Рис. 7.1. Элементы массива в памяти

values [0]	197
values [1]	
values [2]	-101
values [3]	547
values [4]	
values [5]	350
values [6]	
values [7]	
values [8]	
values [9]	35

Рис. 7.2. Значения, которыми инициализирован массив

```
int values[10];
values[0] = 197;
values[2] = -100;
values[5] = 350;
values[3] = values[0] + values[5];
values[9] = values[5] / 10;
--values[2];
```

При первом присваивании элементу массива `values[0]` присваивается значение 197. Аналогично, при втором и третьем присваивании значения -100 и 350 сохраняются в элементах массива `values[2]` и `values[5]` соответственно. В следующем утверждении суммируется содержимое элемента `values[0]` (его значение 197) с содержимым `values[5]` (350) и результат 547 сохраняется в элементе массива `values[3]`. В очередном утверждении элементу массива `values[9]` присваивается значение, полученное в результате деления значения элемента `values[5]` на 10. В последнем утверждении декрементируется значение элемента массива `values[2]`, которое изменяется с -100 на -101.

Ранее рассмотренный фрагмент программы используется в программе из листинга 7.1. В цикл `for` последовательно просматриваются элементы массива и отображаются их значения на терминале.

Листинг 7.1. Работа с массивами

```
#include <stdio.h>
int main (void)
{
    int values[10];
    int index;
    values[0] = 197;
    values[2] = -100;
    values[5] = 350;
    values[3] = values[0] + values[5];
    values[9] =
```

```

values[5] / 10;
--values[2];

for ( index = 0; index < 10; ++index )
    printf ("values[%i] = %i\n", index, values[index]);

return 0;
}
values [0]
values [1]
values [2]
values [3]
values [4]
values [5]
values [6]
values [7]
values [8]
values [9]
197
-101
547
350
35

```

Листинг 7.1. Вывод

```

values[0] = 197
values[1] = 0
values[2] = -101
values[3] = 547
values[4] = 0
values[5] = 350
values[6] = 0
values[7] = 0
values[8] = 0
values[9] = 35

```

Переменная `index` последовательно принимает значения от 0 до 9, т.к. последний элемент должен иметь индекс на единицу меньше, чем количество элементов (вспомните, что отсчет начинается с нулевого элемента). Поскольку не присваиваются значения пяти элементам массива — это элементы 1, 4 и элементы от 6 до 8, то для них нельзя отображать никаких значений. Хотя при выводе эти значения отображаются как нулевые, значения любых неинициализированных переменных считаются неопределенными. По этой причине никаких предположений о значении неинициализированных переменных делать нельзя.

Использование элементов массива в качестве счетчиков

Рассмотрим простой практический пример. Предположим, вы проводите небольшое исследование с целью узнать мнение людей о некоторой телевизионной передаче и просите их оценить передачу по десятибалльной шкале. После опроса 5000 человек вы получите список в состоящий из 5000 значений. После этого необходимо будет

проанализировать результаты. Для этого составляется таблица, отображающая результаты опроса, в которой будет показано, как много людей оценили передачу на 1 балл, как много на 2 балла и так далее до 10.

Хотя это и не самая рутинная операция, но будет довольно утомительно просматривать результаты опроса и вручную вносить данные в таблицу. В нашем случае это всего 10 позиций, но может возникнуть задача провести анализ для определенного возрастного диапазона и делать это вручную будет совсем неразумно. Поэтому необходимо разработать программу для подсчета числа людей по каждому рейтингу. При разработке программы самым очевидным решением может показаться создание 10 отдельных счетчиков, возможно, обозначенных как `rating_1`, `rating_2` и так до 10, и затем инкрементировать значения каждого счетчика при получении соответствующей оценки. Но при этом вы получите довольно неудобную и громоздкую программу. Гораздо удобнее для реализации счетчиков использовать массив.

Например, вы можете создать массив счетчиков с именем `ratingCounters` и инкрементировать каждый элемент массива в соответствии с введенным значением. Из-за ограничений по объему, мы рассмотрим программу только для 20 респондентов (листинг 7.2). Всегда лучше проверить программу на небольшом числе значений, прежде чем использовать полностью весь допустимый диапазон. При этом будет значительно легче обнаружить возможные недоработки.

Листинг 7.2. Демонстрация использования массива счетчиков

```
#include <stdio.h>
int main (void)
{
    int ratingCounters[11], i, response;

    for ( i = 1; i <= 10; ++i )
        ratingCounters[i] = 0;
    printf ("Enter your responses\n");
    for ( i = 1; i <= 20; ++i ) {
        scanf ("%i", &response);
        if ( response < 1 || response > 10 )
            printf ("Bad response: %i\n", response);
        else
            ++ratingCounters[response];
    }
    printf ("\n\nRating Number of Responses\n");
    printf ("-----\n");
    for ( i = 1; i <= 10; ++i )
        printf ("%4i%14i\n", i, ratingCounters[i]);

    return 0;
}
```

Листинг 7.2. Вывод

```
Enter your responses
6
5
8
3
9
6
```

```

5
7
15
Bad response: 15
5
5
1
7
4
10
5
5
6
8
9
Rating Number of Responses
-----
1 1
2 0
3 1
4 1
5 6
6 3
7 2
8 2
9 2
10 1

```

Массив `ratingCounters` объявлен для хранения 11 значений. При этом вы можете возникнуть почему массив имеет 11 элементов, когда всего задано 10 возможных вариантов? Это необходимо сделать для упрощения подсчета значений отдельных рейтингов, поскольку рейтинги могут принимать значения от 1 до 10. Поскольку каждый рейтинг может принимать одно из 10 значений, то в программе просто инкрементируется соответствующий элемент массива, но при этом необходимо сделать проверку, что введен допустимый рейтинг. Например, если введен рейтинг 5, то будет инкрементироваться элемент массива с индексом 5 — `ratingCounters[5]`. После проведения полного подсчета, в элементе массива `ratingCounters[5]` будет содержаться общее количество людей, оценивших передачу на 5.

Поскольку наибольшим значением будет 10, то оно должно быть на единицу меньше размера массива, и при этом мы будем использовать элемент `ratingCounters[10]`, но не элемент с индексом 0, т.к. нет оценки 0. При этом обратите внимание на то, что переменная `i` (счетчик) для цикла `for`, в котором инициализируются и отображаются элементы массива, должна начинаться со значения 1, и таким образом будут использоваться только значения от 1 до 10 и не учитываться элемент `ratingCounters[0]`.

В заключение рассмотрим, как можно использовать массив с 10 элементами для подсчета рейтингов. В этом случае вы должны при вводе рейтинга инкрементировать элемент массива на единицу меньше рейтинга — `ratingCounters[response-1]`. При этом элемент массива `ratingCounters[0]` будет содержать число респондентов, оценивших передачу на 1, элемент `ratingCounters[1]` будет содержать число респондентов, оценивших передачу на 2, и т.д. Это будет вполне приемлемое решение, и единственным его недостатком является возможность внесения путаницы при составлении и чтении программы.

Генерация чисел Фибоначчи

Изучите программу из листинга 7.3, в которой производится генерация первых 15 чисел Фибоначчи, и постарайтесь предсказать результат. Какие взаимозависимости существуют между числами в таблице?

Листинг 7.3. Генерация чисел Фибоначчи

```
// Программа генерирует первые 15 чисел Фибоначчи.
#include <stdio.h>
int main (void)
{
    int Fibonacci[15], i;
    Fibonacci[0] = 0; // По умолчанию.
    Fibonacci[1] = 1; // По умолчанию.

    for ( i = 2; i < 15; ++i )
        Fibonacci[i] = Fibonacci[i-2] + Fibonacci[i-1];
    for ( i = 0; i < 15; ++i )
        printf ("%i\n", Fibonacci[i]);

    return 0;
}
```

Листинг 7.3. Вывод

```
0
1
1
2
3
5
8
13
21
34
55
89
144
233
377
```

Первые два числа Фибоначчи, обозначенные как F_0 и F_1 , объявлены со значениями 0 и 1 соответственно. Каждое последующее число Фибоначчи F_i является суммой двух предыдущих чисел Фибоначчи F_{i-2} и F_{i-1} . Поэтому F_2 является суммой двух значений F_0 и F_1 . В представленной выше программе это достигается непосредственным суммированием значений `Fibonacci[0]` и `Fibonacci[1]`. Это суммирование выполняется в цикле `for`, в котором и рассчитываются все значения чисел Фибоначчи от F_2 до F_{14} (или от `Fibonacci[2]` до `Fibonacci[14]`).

Числа Фибоначчи используются не только во многих областях математики, но и при изучении компьютерных алгоритмов. Последовательность чисел Фибоначчи была получена при изучении т.н. “проблемы кроликов”. Если предположить, что возьмете пару кроликов, то в самом общем случае от этой пары каждый месяц будет рождаться

еще пара кроликов, то соответственно, можно также предположить, что от каждой рожденной пары кроликов каждый второй месяц будет рождаться также по паре кроликов. А если предположить, что кролики не умирают в течение года, то сколько пар кроликов будет получено в конце года? Это количество будет увеличиваться согласно закономерности чисел Фибоначчи, и к концу n месяца будет F_{n+2} кроликов. Поэтому, согласно таблице, полученной в результате выполнения программы, к концу 12-го месяца будет уже 377 пар кроликов.

Использование массивов для генерации простых чисел

Давайте вернемся к программе генерации простых чисел, которая была разработана в главе 6, “Принятие решений”, и проанализируем, как можно использовать массивы при разработке более эффективной программы. В программе из листинга 6.10А для получения простого числа было использовано последовательное деление очередного числа на все последующие целые числа начиная с 2. В упражнении 7 из главы 6 были устранены некоторые логические неточности, присущие этому алгоритму. Но даже после этих изменений данный алгоритм все еще не является самым эффективным. Правда это совсем не заметно в случае с небольшими числами, не больше 50, но ситуация усложнится, когда вы захотите распечатать таблицу простых чисел, например, до одного миллиона.

Для повышения эффективности алгоритма мы используем то свойство простых чисел, что оно не должно делиться без остатка на любое другое простое число. Известно, что любое целое число может быть представлено произведением простых чисел (например, число 20 можно представить как произведение простых чисел 2, 2 и 5). Это свойство можно использовать для разработки эффективного алгоритма получения последовательности простых чисел. Слово “последовательность” должно натолкнуть вас на мысль, что в данном случае можно использовать массивы — для хранения генерируемой последовательности простых чисел.

В целях оптимизации алгоритма также можно использовать тот факт, что любое целое число n может иметь среди своих множителей числа, которые не больше, чем квадратный корень из числа n . Это означает, что проверять множители целого числа можно только до значения, не превышающего квадратного корня из числа.

В программе из листинга 7.4 использованы все улучшения алгоритма, о которых мы только что говорили. Программа генерирует простые числа до значения 50.

Листинг 7.4. Улучшенная программа генерации простых чисел, версия 2

```
#include <stdio.h>
#include <stdbool.h>
// Modified program to generate prime numbers
int main (void)
{
    int p, i, primes[50], primeIndex = 2;
    bool isPrime;
    primes[0] = 2;
    primes[1] = 3;
    for ( p = 5; p <= 50; p = p + 2 ) {
        isPrime = true;
        for ( i = 1; isPrime && p / primes[i] >= primes[i]; ++i )
            if ( p % primes[i] == 0 )
                isPrime = false;
        if ( isPrime == true ) {
            primes[primeIndex] = p;
```

```

++primeIndex;
}
}
for ( i = 0; i < primeIndex; ++i )
printf ("%i ", primes[i]);
printf ("\n");
return 0;
}

```

Листинг 7.4. Вывод

2 3 5 7 11 13 17 19 23 29 31 37 41 43 47

Выражение

```
p / primes[i] >= primes[i]
```

используется во внутреннем цикле `for` для проверки того, что значение переменной `p` не превышает квадратного корня из `primes[i]`. Это мы уже обсуждали в предыдущем разделе, хотя и не углублялись в вопросы математики.

Программа из листинга 7.4 начинается с сохранения чисел 2 и 3 в массиве `primes` как первых двух простых чисел. Массив `primes` объявлен с числом элементов 50, хотя очевидно, что простых чисел до числа 50 будет значительно меньше. Переменная `primeIndex` инициализируется значением 2, которое указывает на очередной незаполненный элемент массива. Затем начинает выполняться цикл `for` для просмотра всех нечетных чисел от 5 до 50. После того как булева переменная `isPrime` примет значение `true`, запускается еще один цикл `for`. В этом цикле производится последовательное деление переменной `p` на все предыдущие простые числа, уже сгенерированные в программе и сохраненные в массиве `primes`. Индексная переменная `i` инициализируется значением 1, поскольку нет необходимости производить проверку деления числа `p` на значение `primes[0]`, которое равно 2. Здесь учитывается то, что в программе проверяются только нечетные числа. Внутри цикла производится проверка деления значения переменной `p` на значения `primes[i]`. Если это условие выполняется, булева переменная `isPrime` принимает значение `false`. Цикл `for` продолжает выполняться до тех пор, пока значением переменной `isPrime` является `true` и значение элемента массива `primes[i]` не превышает квадратного корня из `p`.

После окончания цикла `for` производится проверка значения переменной `isPrime` для определения того, является ли очередное значение переменной `p` простым числом и надо ли его сохранять в массиве `primes`. После того как будут получены все простые числа, программа отображает все значения, сохраненные в массиве `primes`. Значение индексной переменной `i` изменяется от 0 до `primeIndex-1`, поскольку переменная `primeIndex` всегда указывает на очередной свободный элемент массива.

Инициализация массивов

Поскольку можно присвоить значение любой переменной, которая уже объявлена, точно так же можно присвоить значение и элементам массива. Это выполняется путем простого перечисления начальных значений массива начиная с первого элемента. Значения в списке разделяются запятыми, а весь список заключается в фигурные скобки. Объявление

```
int counters[5] = { 0, 0, 0, 0, 0 };
```

вводит массив с именем `counters`, который рассчитан на хранение пяти целых чисел и каждый элемент инициализирован начальным значением 0. Аналогично, в объявлении

```
int integers[5] = { 0, 1, 2, 3, 4 };
```

элементу массива `integers[0]` присвоено значение 0, элементу `integers[1]` присвоено значение 1, `integers[2]` значение 2 и т.д.

Аналогичным образом можно инициализировать и массивы символов. Рассмотрим следующее утверждение

```
char letters[5] = { 'a', 'b', 'c', 'd', 'e' };
```

в котором объявляется массив `letters` и инициализируется пятью символами 'a', 'b', 'c', 'd' и 'e'.

Нет никакой необходимости полностью инициализировать все элементы массива. Если задано несколько инициализирующих значений, то только им и присваиваются начальные значения. Остальные элементы массива устанавливаются в нуль. Поэтому объявление

```
float sample_data[500] = { 100.0, 300.0, 500.5 };
```

инициализирует первые три элемента массива значениями 100.0, 300.0 и 500.5, и устанавливает значения остальных 497 элементов в нуль. Можно написать и следующим образом.

```
float sample_data[500] = { [2] = 500.5, [1] = 300.0, [0] = 100.0 };
```

здесь массив `sample_data` инициализируется теми же самыми значениями, что и в предыдущем случае. А в утверждениях

```
int x = 1233;  
int a[10] = { [9] = x + 1, [2] = 3, [1] = 2, [0] = 1 };
```

объявляется 10-элементный массив и инициализируются первые три элемента значениями 1, 2 и 3, а также последний элемент значением `x+1` (или 1234).

К сожалению, нельзя автоматизировать ввод значений в массив с экрана. Поэтому если вам надо заполнить массив из 500 элементов значением 1, то придется заполнять каждый элемент в отдельности. Разумеется, в этом случае лучше использовать программный цикл `for` для заполнения массива, но не всегда надо будет инициализировать массив только значениями 1.

В программе из листинга 7.5 показаны два способа инициализации массива.

Листинг 7.5. Инициализация массивов

```
#include <stdio.h>  
int main (void)  
{  
    int array_values[10] = { 0, 1, 4, 9, 16 };  
    int i;  
  
    for ( i = 5; i < 10; ++i )  
        array_values[i] = i * i;  
    for ( i = 0; i < 10; ++i )  
        printf ("array_values[%i] = %i\n", i, array_values[i]);  
}
```

```
    return 0;
}
```

Листинг 7.5. Вывод

```
array_values[0] = 0
array_values[1] = 1
array_values[2] = 4
array_values[3] = 9
array_values[4] = 16
array_values[5] = 25
array_values[6] = 36
array_values[7] = 49
array_values[8] = 64
array_values[9] = 81
```

При объявлении массива `array_values`, первые пять элементов массива инициализируются квадратными корнями их порядковых номеров (например, элемент номер 3 принимает значение 3^2 , или 9). В цикле `for` показано, как тоже самое можно сделать внутри цикла. В этом цикле также каждый элемент от 5 до 9 инициализируются квадратными корнями их порядковых номеров. Второй цикл `for` просто выполняет вывод значений всех десяти элементов на дисплей.

Символьные массивы

Целью программы из листинга 7.6 является демонстрация того, как можно использовать символьный массив. Однако некоторые ее моменты достойны отдельного обсуждения.

Листинг 7.6. Введение в символьные массивы

```
#include <stdio.h>
int main (void)
{
    char word[] = { 'H', 'e', 'l', 'l', 'o', '!' };
    int i;

    for ( i = 0; i < 6; ++i )
        printf ("%c", word[i]);
    printf ("\n");

    return 0;
}
```

Листинг 7.6. Вывод

```
Hello!
```

Первый момент, на который необходимо обратить внимание, — это объявление массива `word`. Здесь не указано число элементов массива. В языке программирования C разрешается объявлять массив без указания числа элементов. И если написать именно

так, то размер массива будет определяться автоматически по числу значений для инициализации массива. Поскольку в листинге 7.6 задано шесть значений для инициализации массива `word`, то компилятор языка C присвоит массиву размер в шесть элементов.

Но такой способ хорошо работает, если инициализируется каждый элемент массива. В ином случае необходимо задавать размер массива. Но можно использовать индексы при инициализации массива, как показано ниже.

```
float sample_data[] = { [0] = 1.0, [49] = 100.0, [99] = 200.0 };
```

В этом случае наибольший индекс определяет размер массива. В нашем примере массив `sample_data` будет содержать 100 элементов, т.к. наибольший указанный индекс имеет значение 99.

Преобразования системы счисления с помощью массивов

Следующая программа демонстрирует использование целочисленных и символьных массивов. Эта программа разработана для преобразования положительных целых чисел с основанием 10 в эквивалентные числа с основанием 16. При вводе значений вы указываете само число, которое должно быть преобразовано, и основание для нового числа. После этого программа преобразовывает введенное число и отображает результат с необходимым основанием.

Первым шагом при разработке такой программы будет написание алгоритма для преобразования системы счисления с основанием 10 в другую систему счисления. Алгоритм для генерации цифр преобразованного числа можно неформально выразить следующим образом. Цифры преобразованного числа получаются с помощью деления по модулю исходного числа на новое основание системы счисления. При этом остаток учитывается в качестве новой цифры, а результат деления вновь делится на новое основание и т.д. Процесс повторяется до тех пор, пока результат деления не станет равным нулю.

Результирующие цифры получаются при чтении остатков слева направо. Рассмотрим следующий пример. Предположим, вы хотите преобразовать число 10 в десятичной системе счисления в число в двоичной системе счисления. В таблице приведена последовательность шагов, которые должны быть выполнены при этом преобразовании.

Таблица 7.1. Преобразование целого числа из десятичной системы счисления в двоичную

Число	Остаток	Число / 2
10	0	5
5	1	2
2	0	1
1	1	0

Результатом преобразования числа 10 из десятичной системы счисления в двоичную будет число 1010, которое получается при считывании цифр остатка снизу вверх.

При написании программы, которая выполняет данное преобразование, вы должны учесть некоторые особенности. Во-первых, то, что алгоритм генерирует числа в

обратном порядке, не очень удобно. Нельзя допустить, чтобы пользователь сам переставлял цифры полученного числа. Поэтому прежде чем отобразить результирующее число, его необходимо сохранить в массиве и отобразить в обратном порядке.

Во-вторых, вы должны подумать о том, как отображать числа с основанием 16. В этом случае числа от 10 до 15 должны отображаться с помощью букв от А до F.

Изучите программу из листинга 7.7 и проанализируйте, как решаются эти проблемы. В этой программе также используется спецификатор (уточнитель) `const`, который используется с переменными, значение которых в программе не должно изменяться.

Листинг 7.7. Преобразование целого числа из одной системы счисления в другую

```
// Программа преобразования целого числа из одной системы счисления
в другую
#include <stdio.h>
int main (void)
{
    const char baseDigits[16] = {
        '0', '1', '2', '3', '4', '5', '6', '7',
        '8', '9', 'A', 'B', 'C', 'D', 'E', 'F' };
    int convertedNumber[64];
    long int numberToConvert;
    int nextDigit, base, index = 0;

    // Получить число и основание.
    printf ("Number to be converted? ");
    scanf ("%ld", &numberToConvert);
    printf ("Base? ");
    scanf ("%i", &base);

    // Преобразовать в указанное основание.
    do {
        convertedNumber[index] = numberToConvert % base;
        ++index;
        numberToConvert = numberToConvert / base;
    }
    while ( numberToConvert != 0 );

    // Отобразить результат в обратном порядке.
    printf ("Converted number = ");
    for (--index; index >= 0; --index ) {
        nextDigit = convertedNumber[index];
        printf ("%c", baseDigits[nextDigit]);
    }
    printf ("\n");

    return 0;
}
```

Листинг 7.7. Вывод

```
Number to be converted? 10
Base? 2
Converted number = 1010
```

Листинг 7.7. Вывод (Повторение)

```
Number to be converted? 128362
Base? 16
Converted number = 1F56A
```

Спецификатор const

В языке программирования С при объявлении переменных, которые не должны изменяться в программе, можно использовать спецификатор `const`. Таким образом вы сообщаете компилятору, что эта переменная содержит константное значение, т.е. является константой. Таким переменным нельзя присваивать значения в программе, нельзя их инкрементировать или декрементировать. Если это сделать, то компилятор зафиксирует ошибку и выдаст сообщение о ней. Одной из причин введения спецификатора `const` явилось то, что компилятор может разместить переменную в памяти, предназначенной только для чтения. Обычно все команды программы размещаются в такой памяти.

Как пример использования спецификатора `const`, рассмотрим следующее объявление

```
const double pi = 3.141592654;
```

где переменная `pi` объявляется как константа. Это говорит компилятору о том, что данная переменная не должна изменяться в программе. Если вы случайно напишете утверждение

```
pi = pi / 2;
```

то компилятор `gcc` выдаст следующее предупреждающее сообщение.

```
foo.c:16: warning: assignment of read-only variable 'pi'
```

Возвратимся к программе из листинга 7.7, где символьный массив `baseDigits` рассчитан на хранение 16 символов, которые будут отображаться для преобразованного числа. Массив объявлен со спецификатором `const`, поскольку его содержимое не должно изменяться в программе. Необходимо отметить, что это делает программу более защищенной от ошибок разработчика и способствует лучшему ее пониманию при анализе.

Массив `convertedNumber` рассчитан на хранение 64-разрядных цифр, которые могут быть получены в результате преобразования наибольшего целочисленного типа `long` к наименьшему возможному основанию системы счисления 2. Переменная `numberToConvert` объявлена как тип `long int`, поэтому можно преобразовывать относительно большие числа. Наконец, переменная `base` (содержащая основание системы счисления) и переменная `index` (для индексации массива `convertedNumber`) объявлены как целочисленный тип `int`.

После того как пользователь введет число для преобразования и основание системы счисления (обратите внимание, что процедура `scanf` предназначена для считывания значений типа `long`, о чем говорит символ форматирования `%ld`), в программе происходит переход к циклу `do` для выполнения преобразования. Цикл `do` выбран с той целью, чтобы по крайней мере одна цифра была сохранена в массиве `convertedNumber`, даже если пользователь введет для преобразования число 0.

Внутри цикла производится деление по модулю значения переменной `numberToConvert` на основание системы счисления `base` и определяется следующее делимое.

Остаток сохраняется в массиве `convertedNumber`, а индекс массива инкрементируется. После деления значения переменной `numberToConvert` на значение системы счисления, проверяется условие выхода из цикла. Если значение переменной `numberToConvert` равно 0, то выполнение цикла прекращается, в противном случае цикл повторяется и определяется следующая цифра для преобразованного числа.

Когда цикл `do` заканчивается, значением переменной `index` будет количество цифр преобразованного числа. Поскольку значение этой переменной будет на единицу больше последнего значения, то ее значение должно декрементироваться в последующем цикле `for`. Целью этого цикла является отображение преобразованного числа на экране. В цикле производится просмотр всех цифр переменной `convertedNumber` в обратном порядке для того, чтобы число отображалось правильно.

Каждая цифра из массива `convertedNumber` последовательно присваивается переменной `nextDigit`. Для того чтобы цифры от 10 до 15 корректно отображались как буквы от А до F, используется массив `baseDigits`, символы которого и будут отображаться для соответствующих значений цифр, которые используются в качестве индексов. Для цифр от 0 до 9 массив `baseDigits` содержит символы от '0' до '9', которые отличаются от целочисленных значений от 0 до 9. Для индексов от 10 до 15 в массиве `baseDigits` содержатся символы от 'A' до 'F'. Поэтому если, например, значение переменной будет равно 10, то будет отображен символ, содержащийся в ячейке `baseDigits[10]`, или символ 'A'. А если значение переменной `nextDigit` будет равно 8, то будет отображен символ '8', который и содержится в ячейке `baseDigits[8]`.

Когда значение переменной `index` становится меньше нуля, то цикл `for` заканчивается. При этом на экране будет отображена последовательность цифр нового числа. При изучении программы может возникнуть вопрос: можно ли упростить программу и избежать некоторых промежуточных шагов. Например, можно ли не присваивать значение элемента `convertedNumber[index]` переменной `nextDigit`, а непосредственно использовать это выражение в утверждении `printf`. Другими словами, выражение

```
baseDigits[ convertedNumber[index] ]
```

должно быть вставлено в утверждение `printf` и должен быть получен тот же результат. Разумеется, такое выражение выглядит несколько необычно и его труднее понять, чем используемые в программе эквивалентные утверждения, но это только поначалу. Впоследствии они будут восприниматься вполне естественно.

Необходимо признать, что предыдущая программа требует некоторой доработки. В ней не предусмотрено даже простейших проверок относительно того, например, что значение переменной `base` должно находиться между 2 и 16. И если пользователь в качестве основания системы счисления введет 0, что вполне вероятно, то в цикле `do` произойдет деление на 0, что недопустимо. А если пользователь введет 1 для основания системы счисления, то программа никогда не закончится, т.к. значение переменной `numberToConvert` в цикле никогда не станет равным 0. Если пользователь для основания системы счисления введет число, большее 16, то в программе произойдет выход за пределы диапазона массива `baseDigits`. Это еще один момент, который именно пользователь должен не допустить в программе, поскольку компилятор языка программирования C не производит контроль относительно выхода за пределы диапазона.

В главе 8, "Функции", эта программа будет переписана и все проблемные места будут учтены. А сейчас перейдем к рассмотрению расширенных возможностей массивов.

Многомерные массивы

Все типы массивов, которые мы рассматривали до сих пор, являются линейными массивами. Это означает, что все они использовали только одно измерение. В языке программирования C можно использовать массивы с любым количеством измерений. В этом разделе будут рассмотрены двумерные массивы. Одним из наиболее естественных приложений для использования двумерных массивов являются матрицы. Рассмотрим матрицу 4*5, приведенную в табл. 7.2.

Таблица 7.2. Матрица 4*5

10	5	-3	17	82
9	0	0	8	-7
32	20	1	0	14
0	0	8	7	6

В математике наиболее общим способом обращения к элементам матрицы является использование двух подстрочных индексов. Поэтому если для обозначения матрицы использовать символ M , то ссылаться на элементы матрицы следует с обозначения $M_{i,j}$, где символ i соответствует номеру столбца, а символ j указывает номер строки. При этом диапазон изменения i — от 1 до 4, а диапазон изменения j — от 1 до 5.

Обозначение $M_{3,2}$ используется для обращения к 20-му элементу матрицы, который находится на пересечении 3-й строки и 2-го столбца. Аналогично, для обращения к элементу со значением 6, который находится на пересечении 4-й строки и 5-го столбца, необходимо написать $M_{4,5}$.

В языке программирования C вы можете использовать аналогичное написание для обращения к элементам матрицы двумерного массива. Но поскольку в языке программирования C обычно отсчет начинается с нуля, то первая строка матрицы считается нулевой строкой и первый столбец тоже считается нулевым. В предшествующей матрице будет такое распределение строк и столбцов, как показано в табл. 7.3.

Таблица 7.3. Матрица 4*5 в языке программирования C

Столбец (j)	0	1	2	3	4
Строка (i)					
0	10	5	-3	17	82
1	9	0	0	8	-7
2	32	20	1	0	14
3	0	0	8	7	6

Поскольку в математике используются подстрочные индексы, которые не используются в языке программирования C, эквивалентное обозначение для C будет записываться как $M[i][j]$. При этом не забывайте, что первый индекс ссылается на номер строки, а второй символ указывает номер столбца. Поэтому в утверждении

`sum = M[0][2] + M[2][4];`

выполняется суммирование значения, находящегося на пересечении строки 0 и столбца 2 (-3), и значения, находящегося на пересечении строки 2 и столбца 4 (14). В результате переменной `sum` будет присвоено значение 11.

Двумерный массив объявляется так же, как и одномерный массив. Например, в утверждении

```
int M[4][5];
```

объявлен двумерный массив `M`, состоящий из 4 строк и 5 столбцов, с общим количеством элементов 20. Каждый элемент массива должен содержать целочисленное значение.

Двумерный массив можно инициализировать так же, как и одномерный. При этом последовательно перечисляются значения для инициализации. Фигурные скобки необходимы для выделения значений, принадлежащих одной строке. Таким образом, объявление и инициализацию элементов массива `M`, который представлен в таблице 7.3, можно выполнить следующим образом.

```
int M[4][5] = {  
    { 10, 5, -3, 17, 82 },  
    { 9, 0, 0, 8, -7 },  
    { 32, 20, 1, 0, 14 },  
    { 0, 0, 8, 7, 6 }  
};
```

Обратите внимание на синтаксис предыдущего утверждения. Запятые ставятся после фигурной скобки, ограничивающей значения для одной строки, исключая последнюю строку. Хотя использование пар внутренних фигурных скобок не является обязательным. Они поставлены для того, чтобы инициализировать значения по строкам. Это же самое утверждение можно написать и следующим образом.

```
int M[4][5] = { 10, 5, -3, 17, 82, 9, 0, 0, 8, -7, 32, 20, 1, 0, 14,  
0, 0, 8, 7, 6 };
```

Как и при инициализации одномерного массива, вовсе не требуется задавать все значения для инициализации всех элементов. При таком объявлении, как

```
int M[4][5] = {  
    { 10, 5, -3 },  
    { 9, 0, 0 },  
    { 32, 20, 1 },  
    { 0, 0, 8 }  
};
```

будут проинициализированы только по три элемента для каждой строки матрицы. Остальные элементы матрицы примут значение 0. Обратите внимание, что в этом случае внутренние фигурные скобки, выделяющие строки, уже необходимы. Если их не поставить, то будут инициализированы первые две строки и первые два элемента третьей строки.

Индексы также можно использовать при инициализации, как и в случае с одномерным массивом. Поэтому объявление

```
int matrix[4][3] = { [0][0] = 1, [1][1] = 5, [2][2] = 9 };
```

инициализирует три указанных элемента матрицы `matrix`. Неуказанные элементы будут инициализированы нулевым значением по умолчанию.

Массивы с переменной длиной

В этом разделе обсуждаются возможности языка по работе с массивами без задания им постоянных размеров.

Во всех ранее представленных примерах этой главы при объявлении указывался размер массива. В языке программирования C разрешается объявлять массив без указания размера. Например, в листинге 7.3 приведена программа расчета первых 15 чисел Фибоначчи. А если вы захотите рассчитать числа Фибоначчи для 100 или даже 500 значений? А возможно, вы захотите, чтобы пользователь задавал необходимое количество значений для генерации чисел Фибоначчи. Изучите программу 7.3 и проанализируйте, как в ней решены эти проблемы.

Не все компиляторы поддерживают возможность использования массивов с неопределенным размером. Ознакомьтесь с документацией по компилятору, прежде чем использовать массивы с переменным размером.

Листинг 7.8. Генерация чисел Фибоначчи с помощью массивов с переменным размером

```
// Генерация чисел Фибоначчи с помощью массивов с переменным размером
#include <stdio.h>
int main (void)
{
    int i, numFibs;

    printf ("How many Fibonacci numbers do you want (between 1 and 75)? ");
    scanf ("%i", &numFibs);
    if (numFibs < 1 || numFibs > 75) {
        printf ("Bad number, sorry!\n");
        return 1;
    }
    unsigned long long int Fibonacci[numFibs];
    Fibonacci[0] = 0; // По умолчанию.
    Fibonacci[1] = 1; // По умолчанию.
    for ( i = 2; i < numFibs; ++i )
        Fibonacci[i] = Fibonacci[i-2] + Fibonacci[i-1];
    for ( i = 0; i < numFibs; ++i )
        printf ("%llu ", Fibonacci[i]);
    printf ("\n");

    return 0;
}
```

Листинг 7.8. Вывод

```
How many Fibonacci numbers do you want (between 1 and 75)? 50
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584
4181 6765 10946 17711 28657 46368 75025 121393 196418 317811 514229
832040 1346269 2178309 3524578 5702887 9227465 14930352 24157817
39088169 63245986 102334155 165580141 267914296 433494437 701408733
1134903170 1836311903 2971215073 4807526976 7778742049
```

Программа из листинга 7.8 содержит несколько фрагментов, которые необходимо обсудить. Во-первых, объявлены переменные `i` и `numFibs`. Эти переменные используются для хранения требуемого количества чисел Фибоначчи, которые пользователь хочет сгенерировать. Обратите внимание, что введенное число подвергается проверке на предмет выхода за пределы диапазона, что является примером хорошего стиля программирования. Если введенное значение выходит за пределы диапазона, т.е. оно меньше 1 или больше 75, то отображается соответствующее сообщение и происходит выход из программы со значением 1.

Выполнение утверждения `return` в этом месте приводит к немедленному выходу из программы без выполнения всех оставшихся утверждений. Как уже отмечалось в главе 3, “Компиляция и запуск первой программы”, если возвращается ненулевое значение, то обычно это свидетельствует об аварийном выходе из программы, а вызывающая программа может проанализировать возвращаемое значение, если в ней это предусмотрено.

После введения пользователем числа появляется следующее утверждение.

```
unsigned long long int Fibonacci[numFibs];
```

Массив `Fibonacci` объявлен с размером `numFibs`. То есть размер массива определяет переменная `numFibs`, которая во время выполнения может принимать различные значения. Поэтому размер массива не является жестко заданным во время компиляции. Как уже отмечалось, переменная может быть объявлена в любом месте программы, поэтому объявление можно сделать непосредственно перед утверждением, где переменная будет использоваться. И хотя это не приветствуется многими программистами, такое размещение объявлений является вполне законным. Однако, по негласному соглашению, обычно все объявления группируют в одном месте, что значительно удобнее для чтения и способствует ее пониманию.

Поскольку числа Фибоначчи возрастают очень быстро, тип массива объявлен для хранения наибольших значений, которые только можно задать, т.е. `unsigned long long int`. В качестве упражнения вы можете рассчитать наибольшее число Фибоначчи, которое можно сохранить в переменной типа `unsigned long long int`.

Остальная часть программы очевидна. Рассчитывается требуемое количество чисел Фибоначчи, которые отображаются на экране, после чего выполнение программы завершается.

Методика, известная как *динамическое выделение памяти*, часто используется для выделения места для массивов во время выполнения программы. Это делается с помощью функций, таких как `malloc` и `calloc`, которые входят в состав стандартной библиотеки языка программирования C. Эти методы подробно рассмотрены в главе 17, “Дополнения и расширенные возможности”.

Как вы могли убедиться, массив является довольно мощной конструкцией, которая используется во всех языках программирования. Примеры программ с использованием многомерных массивов будут представлены в главе 8, которая начинается с детального обсуждения одного из наиболее важных понятий языка программирования C — *функций*.

Упражнения

1. Наберите и выполните все восемь программ, представленных в данной главе. Сравните вывод, сделанный вашей программой, с выводом, приведенным в книге для каждой программы.
2. Модифицируйте программу из листинга 7.1 таким образом, чтобы каждый элемент массива `values` был инициализирован значением 0. Используйте цикл `for` для этой операции.
3. В программе из листинга 7.2 разрешено получить только 20 значений. Исправьте эту программу таким образом, чтобы можно было получить любое количество значений. Для того чтобы пользователь мог не задавать количество значений, используйте значение 999 для индикации того, что введено последнее значение. (Подсказка: можно использовать утверждение `break` в том месте, где вы выполняете выход из цикла.)
4. Напишите программу, в которой производится расчет среднего значения для массива из 10 вещественных чисел.
5. Какой вывод будет сделан следующей программой?

```
#include <stdio.h>
int main (void)
{
    int numbers[10] = { 1, 0, 0, 0, 0, 0, 0, 0, 0, 0 };
    int i, j;

    for ( j = 0; j < 10; ++j )
        for ( i = 0; i < j; ++i )
            numbers[j] += numbers[i];

    for ( j = 0; j < 10; ++j )
        printf ("%i ", numbers[j]);
    printf ("\n");

    return 0;
}
```

6. Массивы для генерации чисел Фибоначчи нет никакой необходимости использовать. Вы можете просто использовать три переменные: две для хранения предыдущих значений чисел Фибоначчи и одну для хранения текущего значения. Перепишите программу из листинга 7.3 таким образом, чтобы не использовать массив. Поскольку уже не будет массива чисел, вы должны генерировать значение числа Фибоначчи при каждой итерации цикла.
7. Простые числа можно получить с помощью алгоритма, известного как *решето Эратосфена*. Описание этого алгоритма приведено ниже. Напишите программу, которая реализует данный алгоритм. С помощью программы рассчитайте все простые числа до $n=150$. Сравните этот алгоритм с тем, который приведен в главе для расчета простых чисел.

Решето Эратосфена. Алгоритм.

Отображение всех простых чисел от 1 до n .

- Шаг 1.** Задайте массив целых чисел P . Инициализируйте элементы массива P_i значением 0, где $2 \leq i \leq n$.
- Шаг 2.** Присвойте переменной i значение 2.
- Шаг 3.** Если $i > n$, выполнение прекращается.
- Шаг 4.** Если P_i равно 0, то i является простым числом.
- Шаг 5.** Для всех положительных целых значений переменной j , таких как $i * j \leq n$, присваивайте ячейке P_{i*j} значение 1.
- Шаг 6.** Инкрементируйте i и перейдите к шагу 3.

Функции

В основе всех программ на языке программирования C лежат одни и те же фундаментальные элементы — функции. Вы пользуетесь функциями во всех программах, с которыми вы работаете. Подпрограммы `printf` и `scanf` являются примерами функций. А функция `main` является обязательной для любой программы. Вы можете спросить: а можно ли обойтись без функций? Ответ будет однозначный — нельзя. Функции обеспечивают механизм создания программ, которые легко писать, читать, понимать, отлаживать, модифицировать и устанавливать. Как видим, все, что связано с функциями, требует самого пристального внимания.

Объявление функций

Во-первых, следует очень хорошо уяснить, что собой представляют функции, и только после этого вы сможете эффективно их использовать при разработке своих программ. Возвратимся к первой программе, которую вы написали (см. листинг 3.1) и которая отображает на терминале фразу: “Programming is fun.” (Программирование — это забавно).

```
#include <stdio.h>
int main (void)
{
    printf ("Programming is fun.\n");

    return 0;
}
```

Ниже приведена функция, которая делает то же самое.

```
void printMessage (void)
{
    printf ("Programming is fun.\n");
}
```

Различие между функцией `printMessage` и функцией `main` из листинга 3.1 касается первой и последней строки. Первая строка в объявлении функции сообщает компилятору четыре вещи о функции (при чтении слева направо).

1. Кто может ее вызвать (обсуждается в главе 15, “Большие программы”);
2. Тип возвращаемого значения;
3. Название функции;
4. Принимаемые аргументы.

Из первой строки объявления функции `printMessage` компилятор может выяснить, что эта функция не возвращает никакого значения (об этом говорит первое ключевое слово `void`), что имя этой функции `printMessage` и что она не принимает никаких аргументов (об этом говорит второе ключевое слово `void`). Скоро вы узнаете несколько больше о ключевом слове `void`.

Очевидно, что необходимо внимательно отнестись к выбору имени функции — оно должно быть понятно и неоднозначно передавать смысл выполняемых функцией задач. Как и использование понятных идентификаторов, это значительно повышает восприятие логики программы. Вернемся к обсуждению программы из листинга 3.1, в котором используется функция с именем `main`. Это имя в языке программирования C имеет специальное назначение — именно с этой функции начинается выполнение программы. В программе всегда должна присутствовать функция с именем `main`. Функция с этим именем может находиться в любом месте программы и даже завершить программу, как показано в листинге 8.1.

Листинг 8.1. Создание программы на языке программирования C

```
#include <stdio.h>
void printMessage (void)
{
    printf ("Programming is fun.\n");
}
int main (void)
{
    printMessage ();

    return 0;
}
```

Листинг 8.1. Вывод

```
Programming is fun.
```

Программа из листинга 8.1 состоит из двух функций: `printMessage` и `main`. Выполнение программы всегда начинается с функции `main`. Внутри этой функции находится утверждение `printMessage ()`; , которое говорит о том что должна выполняться функция `printMessage`. Открывающая и закрывающая скобки используются для того, чтобы сообщить компилятору, что идентификатор `printMessage` обозначает функцию и что в данном случае в функцию не передается никаких аргументов (таким образом функции всегда задаются в программе).

Когда происходит вызов функции, выполнение программы передается непосредственно вызванной функции. Внутри функции `printMessage` выполняется утверждение `printf` для отображения на экране сообщения “Программирование забавно”. После того как сообщение будет отображено, выполнение подпрограммы `printMessage` заканчивается (об этом говорит закрывающая фигурная скобка) и управление передается в процедуру `main`, где выполнение продолжается с той точки, в которой

закончился вызов функции. Обратите внимание, что желательно поставить утверждение возврата после вызова функции `printMessage`.

```
return;
```

Поскольку функция `printMessage` не возвращает никакого значения, значение для возврата не определено. Использование утверждения возврата является дополнительным, поскольку достижение конца процедуры при выполнении без утверждения возврата приводит к обязательному выполнению всей функции. Использование функции `printMessage` без учета значения возврата аналогично поведению функции без использования утверждения `return`. Подпрограммы `printf` и `scanf` тоже являются функциями. Отличительной особенностью этих функций является то, что они написаны не пользователем, а являются частью стандартной библиотеки функций языка C.

Когда вы используете функцию `printf` для отображения некоторых сообщений или результатов вычислений, выполнение передается функции `printf`, которая выполняет задание и затем передает управление вызвавшей ее процедуре. В любом случае, управление передается непосредственно следующему за вызовом функции утверждению.

Постарайтесь предсказать результат работы программы из листинга 8.2.

Листинг 8.2. Вызовы функций

```
#include <stdio.h>
void printMessage (void)
{
    printf ("Programming is fun.\n");
}
int main (void)
{
    printMessage ();
    printMessage ();

    return 0;
}
```

Листинг 8.2. Вывод

```
Programming is fun.
Programming is fun.
```

Выполнение программы начинается с функции `main`, в которой содержатся два вызова функции `printMessage`. Когда производится вызов функции `printMessage`, управление передается этой функции, что приводит к отображению на дисплее сообщения "Programming is fun". Затем управление передается основной процедуре. После возврата к основной процедуре происходит второй вызов функции `printMessage`, выполнение которой приводит к тому же результату, что и в первый раз. После выполнения этой функции управление опять передается основной программе и выполнение всей программы завершается.

Заканчивая работу с функцией `printMessage`, постарайтесь предсказать результат работы программы из листинга 8.3.

Листинг 8.3. Дополнительные примеры вызова функций

```
#include <stdio.h>
void printMessage (void)
{
    printf ("Programming is fun.\n");
}
int main (void)
{
    int i;
    for ( i = 1; i <= 5; ++i )
        printMessage ();

    return 0;
}
```

Листинг 8.3 Вывод

```
Programming is fun.
Programming is fun.
Programming is fun.
Programming is fun.
Programming is fun.
```

Аргументы и локальные переменные

Когда вы вызываете функцию `printf`, вы всегда передаете в нее одно или несколько значений. Первое значение обычно является строкой форматирования, а остальные значения являются специфическими величинами, которые вы хотите отобразить на экране. Эти значения называются *аргументами* и они значительно повышают эффективность и гибкость использования функций. В отличие от функции `printMessage`, вызов которой приводит к отображению только одной и той же фразы, вызов функции `printf` используется для отображения заданных вами значений.

Вы сами можете объявить функцию, которая принимает аргументы. В главе 5, “Программные циклы”, вы разрабатывали программы для расчета треугольных чисел. Далее вы должны объявить функцию, которая генерирует треугольные числа. Название ее выберем таким, чтобы был понятен смысл выполняемой функцией задачи: `calculateTriangularNumber`. В качестве аргумента для этой функции будет задан номер треугольного числа, после чего произойдет расчет треугольного числа и отображение результата на экране. В программе из листинга 8.4 демонстрируется использование функции для расчета треугольного числа.

Листинг 8.4. Расчет треугольного числа

```
// Функция для расчета треугольного числа
#include <stdio.h>
void calculateTriangularNumber (int n)
{
    int i, triangularNumber = 0;
    for ( i = 1; i <= n; ++i )
        triangularNumber += i;
    printf ("Triangular number %i is %i\n", n, triangularNumber);
}
```

```
int main (void)
{
    calculateTriangularNumber (10);
    calculateTriangularNumber (20);
    calculateTriangularNumber (50);

    return 0;
}
```

Листинг 8.4. Вывод

```
Triangular number 10 is 55
Triangular number 20 is 210
Triangular number 50 is 1275
```

Объявление прототипа функции

Сделаем несколько пояснений относительно функции `calculateTriangularNumber`. Первая строка функции

```
void calculateTriangularNumber (int n)
```

является *объявлением прототипа функции*. Именно благодаря ей компилятор узнает, что `calculateTriangularNumber` является функцией, которая не возвращает значений (ключевое слово `void`), и что эта функция принимает один аргумент с именем `n`, который имеет тип `int`. Имя, которое выбрано для аргумента, называется *именем формального параметра* и, аналогично имени функции, может быть любым допустимым идентификатором, написанным в соответствии с правилами, описанными в главе 4, “Переменные, типы данных и арифметические выражения”. По вполне очевидной причине это должны быть осмысленные имена.

После того как имена формальных параметров указаны, их можно использовать для ссылки на аргументы в теле функции. Начало описания функции задается открывающей фигурной скобкой. Поскольку в нашем случае необходимо рассчитать треугольное число, то следует задать переменную, которая будет хранить это треугольное число. Также необходимо объявить переменную, которая будет использоваться в качестве счетчика цикла. Переменные `triangularNumber` и `i` объявлены именно для этих целей и имеют тип `int`. Эти переменные объявляются и инициализируются так же, как и переменные внутри основной процедуры `main`, которые мы уже объявляли ранее.

Автоматические локальные переменные

Переменные, объявленные внутри функции, называются *автоматическими локальными переменными*, поскольку они создаются каждый раз при вызове функции и их значения локальны для данной функции, т.е. их можно использовать только в пределах функции. К локальным переменным нельзя обратиться из других функций. Если начальное значение локальной переменной присваивается внутри функции, то это значение будет присваиваться переменной при каждом вызове функции.

Когда объявляется локальная переменная внутри функции, то для уточнения в языке программирования C используется ключевое слово `auto`, которое ставится перед объявлением переменной.

```
auto int i, triangularNumber = 0;
```

Поскольку в языке программирования С компилятор по умолчанию предполагает, что все переменные, объявленные внутри функции, являются автоматическими локальными переменными, ключевое слово `auto` используется редко и по этой причине оно не применяется в данной книге.

Вернемся к программе, где после объявления локальных переменных производится расчет треугольных чисел и отображение результата на экране. Закрывающая фигурная скобка говорит об окончании функции. Внутри процедуры `main` значение 10 передается как аргумент при первом вызове функции `calculateTriangularNumber`. Затем управление передается непосредственно функции, где число 10 становится значением формального параметра `n` внутри функции. Затем функция выполняет расчет 10-го треугольного числа и отображает результат на экране.

Когда производится вызов функции во второй раз, то в качестве аргумента передается значение 20. Как уже было сказано, это число становится значением переменной `n` внутри функции. При этом функция производит расчет 20-го треугольного числа и отображает результат на экране.

В целях демонстрации того, что функция принимает более одного аргумента, перепишем программу определения наибольшего общего делителя (см. листинг 5.7) в виде функции. Двумя аргументами для функции будут являться числа, наибольший общий делитель (`gcd`) которых вы хотите найти. Программа представлена в листинге 8.5.

Листинг 8.5. Новая версия программы определения наибольшего общего делителя

```
/* Функция определения наибольшего общего делителя
   для двух положительных целых чисел */
#include <stdio.h>
void gcd (int u, int v)
{
    int temp;
    printf ("The gcd of %i and %i is ", u, v);
    while ( v != 0 ) {
        temp = u % v;
        u = v;
        v = temp;
    }
    printf ("%i\n", u);
}

int main (void)
{
    gcd (150, 35);
    gcd (1026, 405);
    gcd (83, 240);

    return 0;
}
```

Листинг 8.5. Вывод

```
The gcd of 150 and 35 is 5
The gcd of 1026 and 405 is 27
The gcd of 83 and 240 is 1
```

Функция `gcd` объявлена для работы с двумя целочисленными аргументами. Функция ссылается на эти аргументы через формальные параметры с именами `u` и `v`. После объявления переменной `temp` типа `int`, программа отображает значения параметров `u` и `v` с соответствующим сообщением. Затем в функции производится расчет и отображается наибольший общий делитель для этих значений.

Возможно, вас удивит то, что в функции используются два утверждения `printf`. Но значения переменных `u` и `v` должны быть отображены до начала цикла, т.к. их значения будут изменены в цикле. И если производить их отображение на экран после окончания цикла, они не будут иметь исходных значений. Если вы все же хотите использовать одно утверждение `printf`, то необходимо запомнить начальные значения переменных `u` и `v` во временных переменных, значения которых не изменятся и после окончания цикла. Эти значения могут быть отображены вместе со значением переменной `u` (наибольший общий делитель) с помощью одного утверждения `printf` после окончания цикла.

Возврат результатов работы функции

Функции из листингов 8.4 и 8.5 непосредственно выполняют несколько вычислений и отображают результат на экране. Однако не всегда требуется, чтобы результат вычислений сразу отображался на экране. В языке программирования C применен удобный механизм для возврата результата вычислений в вызывающую программу. Он должен быть вам знаком, т.к. ранее мы уже использовали возвраты из функции `main`. Общий синтаксис для возврата значений достаточно простой.

```
return expression;
```

В этом утверждении указывается на то, что функция должна вернуть значение, полученное в результате расчета выражения `expression`, в вызывающую программу. Круглые скобки, в которые некоторые программисты заключают выражение, являются дополнительными и используются только для стилистической выразительности.

Но только одного утверждения `return` недостаточно. При объявлении функции вы также должны объявить *тип возвращаемого значения*. Это объявление помещается непосредственно перед именем функции. В каждом из предыдущих примеров из этой книги все функции `main` возвращали целочисленное значение, поскольку при объявлении функции ключевое слово `int` помещалось непосредственно перед именем функции. С другой стороны, объявление функции, подобное представленному ниже

```
float kmh_to_mph (float km_speed)
```

приводит к созданию функции с именем `kmh_to_mph`, которая принимает один аргумент типа `float`, для чего объявлен параметр `km_speed`, и возвращает значение типа `float`, о чем говорит ключевое слово `float` в начале объявления. Аналогично, объявление

```
int gcd (int u, int v)
```

определяет функцию с именем `gcd`, которая принимает два целочисленных аргумента и возвращает целочисленное значение. Программа из листинга 8.6 аналогична программе из листинга 8.5, но в ней наибольший общий делитель не отображается в функции `gcd`, а его значение возвращается из функции и отображается в основной программе `main`.

Листинг 8.6. Определение наибольшего общего делителя и возврат результата

```
/* Function to find the greatest common divisor of two
nonnegative integer values and to return the result */
#include <stdio.h>
int gcd (int u, int v)
{
    int temp;
    while ( v != 0 ) {
        temp = u % v;
        u = v;
        v = temp;
    }
    return u;
}

int main (void)
{
    int result;

    result = gcd (150, 35);
    printf ("The gcd of 150 and 35 is %i\n", result);

    result = gcd (1026, 405);
    printf ("The gcd of 1026 and 405 is %i\n", result);

    printf ("The gcd of 83 and 240 is %i\n", gcd (83, 240));

    return 0;
}
```

Листинг 8.6. Вывод

```
The gcd of 150 and 35 is 5
The gcd of 1026 and 405 is 27
The gcd of 83 and 240 is 1
```

После того как значение наибольшего общего делителя вычисляется в функции gcd, для его возврата используется утверждение `return u;`. При этом в вызывающую функцию передается значение переменной `u`, которое является наибольшим общим делителем.

Если вам не совсем понятно, как можно применить возвращаемое из вызванной функции значение внимательно, проанализируйте два первых случая, где возвращаемые значения сохраняются в переменной `result`. Для этого использовано утверждение

```
result = gcd (150, 35);
```

которое означает, что функция принимает два аргумента, 150 и 35, и сохраняет значение, полученное в результате расчета, в переменной `result`.

Результат, который возвращается функцией, вовсе необязательно нужно присваивать переменной, что подтверждается приведенной программой. В последнем случае результат, возвращаемый функцией,

```
gcd (83, 240)
```

передается непосредственно в функцию `printf`, где это значение и выводится на экран.

Как только что было показано, в языке программирования С функция может возвращать только одно значение. В отличие от некоторых других языков программирования, в С нет различия между подпрограммами (процедурами) и функциями. В языке программирования С используются только функции, которые могут возвращать дополнительные значения. Если при объявлении функции пропущен тип возвращаемого значения, компилятор языка С предполагает, что функция возвращает целочисленное значение, если она вообще что-то возвращает. Поэтому некоторые программисты стараются использовать этот факт и не пишут при объявлении тип возвращаемого значения, хотя и используют целочисленное возвращаемое значение, что считается плохим стилем программирования, которого следует избегать.

Если функция должна возвращать значение, то обязательно объявите тип возвращаемого значения при объявлении функции, что улучшит понимание программы. Благодаря этому, читая объявление функции, вы всегда можете узнать не только имя функции и используемые параметры, но и то, что функция возвращает значение и тип этого значения.

Как уже упоминалось, при объявлении функции используется ключевое слово `void`, чтобы подчеркнуть, что функция не возвращает никакого значения. При этом если в программе производится попытка использовать значение, возвращаемое функцией, то возникает ошибка компиляции. Например, функция `calculateTriangularNumber` из листинга 8.4 не возвращает никакого значения, и при ее объявлении использовано ключевое слово `void`, помещенное перед именем функции. Если попытаться использовать возвращаемое значение, например, в утверждении `number=calculateTriangularNumber(20);`, то это приведет к ошибке при компиляции.

В некотором смысле, тип данных `void` на самом деле говорит об отсутствии данных. Поэтому функция, объявленная с типом возвращаемого значения `void`, не может возвращать никакого значения, даже если в ней указано возвращаемое значение.

В главе 6, "Принятие решений", была представлена программа для расчета и отображения абсолютного значения числа. Нам предстоит написать функцию, которая принимает некоторое значение и возвращает результат. В отличие от программы из листинга 6.1, в этом случае функция будет и принимать, и возвращать вещественные значения. Программа представлена в листинге 8.7.

Листинг 8.7. Расчет абсолютного значения

```
// Function to calculate the absolute value
#include <stdio.h>
float absoluteValue (float x)
{
    if ( x < 0 )
        x = -x;

    return x;
}

int main (void)
{
    float f1 = -15.5, f2 = 20.0, f3 = -5.0;
    int i1 = -716;
```



```

float result;

result = absoluteValue (f1);
printf ("result = %.2f\n", result);
printf ("f1 = %.2f\n", f1);

result = absoluteValue (f2) + absoluteValue (f3);
printf ("result = %.2f\n", result);

result = absoluteValue ( (float) i1 );
printf ("result = %.2f\n", result);

result = absoluteValue (i1);
printf ("result = %.2f\n", result);

printf ("%2f\n", absoluteValue (-6.0) / 4 );

return 0;
}

```

Листинг 8.7. Вывод

```

result = 15.50
f1 = -15.50
result = 25.00
result = 716.00
result = 716.00
1.50

```

Функция `absoluteValue` довольно простая. Формальный параметр с именем `x` сравнивается с нулем. Если его значение меньше нуля, то оно преобразовывается в положительное значение для получения абсолютного значения. Результат возвращается в вызывающую программу с помощью ключевого слова `return`.

Необходимо отметить несколько моментов в функции `main`, с помощью которой тестируется функция `absoluteValue`. При первом вызове функции, в нее передается значение переменной `f1`, равное `-15.5`, которое ей присвоено при инициализации. Внутри функции это значение присваивается переменной `x`. Поскольку результат сравнения в нуле будет иметь значение `TRUE`, то выполняется операция инвертирования, в результате которой значение переменной `x` станет равным `15.5`. В следующем утверждении значение переменной `x` возвращается в функцию `main`, где оно присваивается переменной `result` и затем отображается на экране.

Когда значение переменной `x` изменяется внутри функции `absoluteValue`, это никоим образом не сказывается на переменной `f1`. Когда переменная `f1` передается в функцию `absoluteValue`, ее значение автоматически копируется в формальный параметр `x`. Поэтому все изменения, которые производятся со значением, присвоенным переменной `x`, связаны только с переменной `x` и не влияют на переменную `f1`. Это подтверждается во втором утверждении `printf`, с помощью которого отображается значение переменной `f1`, которое не изменилось. Вы должны хорошо понимать, что все изменения, которые производятся внутри функции, влияют только на значения копий, сделанных с аргументов.

В следующих двух вызовах функции `absoluteValue` демонстрируется, как возвращаемое значение может использоваться в арифметических выражениях. Абсолютное

значение переменной `f2` суммируется с абсолютным значением переменной `f3` и сумма присваивается переменной `result`.

В четвертом вызове функции `absoluteValue` показано, что тип аргумента, передаваемого в функцию, должен соответствовать типу параметра, который объявлен внутри функции. Поскольку функция `absoluteValue` в качестве аргумента ожидает тип `float`, то целочисленное значение переменной `i1` сначала приводится к типу `float` и только затем передается в функцию. Но можно и пропустить оператор приведения, т.к. компилятор вполне может определить, какой тип требуется, и самостоятельно сделать приведение. Такая проверка сделана в пятом вызове функции `absoluteValue`. Однако вам должно быть совершенно понятно, что всегда лучше использовать явное приведение типов, чем надеяться, что компилятор сделает это за вас.

В последнем вызове функции `absoluteValue` можно видеть, как производятся вычисления арифметических выражений при различных типах отдельных значений. Поскольку значение, возвращаемое функцией `absoluteValue`, будет иметь тип `float`, то компилятор трактует операцию деления как деление вещественного числа на целое число. Как известно, если одна из переменных, используемых при делении, имеет тип `float`, то деление выполняется по правилам вещественной арифметики. В соответствии с этими правилами, в результате деления вещественного значения -6.0 на целое число 4 получается вещественное значение 1.5 .

Теперь, когда вы разработали функцию подсчета абсолютного значения числа, вы можете ее использовать в своих будущих программах, когда потребуется выполнить такое преобразование. В следующей программе (листинг 8.8) вам предстоит использовать эту функцию.

Функции вызывают функции

При проведении расчетов с использованием современных вычислительных средств, не представляет большого труда найти квадратный корень числа. Но еще несколько лет назад студентов обучали выполнять такие вычисления вручную. Одним из методов, которые используются при подобных вычислениях, был метод итераций Ньютона-Рафсона. В листинге 8.8 представлена программа, в которой используется этот итерационный метод для последовательного приближения результата вычислений к истинному значению.

Метод Ньютона-Рафсона можно описать следующим образом. Вы начинаете с приблизительной оценки квадратного корня числа. Чем точнее будет эта приблизительная оценка, тем меньше будет вычислений для получения истинного значения квадратного корня. Но в нашем случае мы не будем делать приблизительную оценку, а будем предполагать, что приблизительное значение равно 1 .

Затем число, квадратный корень которого вы хотите получить, делится на приблизительное значение и результат добавляется к приблизительному значению. Это промежуточное значение делится на 2 и результат деления становится новым приблизительным значением, которое используется в новом цикле вычислений.

Таким образом, число, квадратный корень которого вы хотите получить, делится на новое приблизительное значение. Результат деления добавляется к приблизительному значению и все это делится на 2 . При этом получается новое приблизительное значение и продолжается итерационный процесс.

Поскольку вы не хотите продолжать этот процесс бесконечно, необходим некоторый критерий для прекращения вычислений. Так как приблизительное значение, которое вы получаете в процессе итераций, становится все ближе и ближе к истинному значению, то можно ввести предел точности, при достижении которого можно

прекратить вычисления. Предел точности обычно обозначают символом эпсилон (ϵ), и если различие между двумя приближительными значениями меньше, чем эпсилон, то вычисления можно прекратить.

Приведенное выше описание можно выразить алгоритмически, как показано ниже.

Метод Ньютона–Рафсона для вычисления квадратного корня числа (x)

Шаг 2. Выбрать приближительное значение 1.

Шаг 2. Если $|\text{guess}2 - x| < \epsilon$, перейти к шагу 4.

Шаг 3. Установить приближительное значение, равное $(x/\text{guess} + \text{guess})/2$, и перейти к шагу 2.

Шаг 4. Считать приближительное значение квадратным корнем числа.

Здесь необходимо сравнивать абсолютное значение разности приближительного значения и числа x со значением допустимой точности, поскольку приближительное значение может быть квадратным корнем числа. Поскольку алгоритм уже разработан, можно переходить к написанию функции расчета квадратного корня числа. В качестве значения для предела точности ϵ в нашем случае выбрано число 0.00001. Сама программа приведена в листинге 8.8

Листинг 8.8. Вычисление квадратного корня числа

```
// Функция для расчета абсолютного значения числа
#include <stdio.h>
float absoluteValue (float x)
{
    if ( x < 0 )
        x = -x;

    return (x);
}

// Функция для расчета квадратного корня числа.
float squareRoot (float x)
{
    const float epsilon = .00001;
    float guess = 1.0;
    while ( absoluteValue (guess * guess - x) >= epsilon )
        guess = ( x / guess + guess ) / 2.0;

    return guess;
}

int main (void)
{
    printf ("squareRoot (2.0) = %f\n", squareRoot (2.0));
    printf ("squareRoot (144.0) = %f\n", squareRoot (144.0));
    printf ("squareRoot (17.5) = %f\n", squareRoot (17.5));

    return 0;
}
```

Листинг 8.8. Вывод

```
squareRoot (2.0) = 1.414216  
squareRoot (144.0) = 12.000000  
squareRoot (17.5) = 4.183300
```

Действительные значения, которые будут отображаться на вашем компьютере, могут незначительно отличаться от приведенных значений. Приведем последовательное описание этой программы. Сначала описывается функция для получения абсолютного значения числа. Эта функция ничем не отличается от той, что приведена в листинге 8.7.

Затем описывается функция для расчета квадратного корня `squareRoot`. В этой функции используется один аргумент с именем `x` и она возвращает значение типа `float`. Внутри тела функции объявлены две локальные переменные с именами `epsilon` и `guess`. Значением переменной `epsilon`, которая используется как предел точности для выхода из итераций, выбрано `0.00001`. Переменная `guess`, которая используется как приблизительное значение квадратного корня числа, инициализируется значением `1.0`. Эти начальные значения присваиваются переменным при каждом вызове функции.

После объявления переменных в листинге находится цикл для выполнения итерационных вычислений. Цикл начинается с ключевого слова `while` и в нем производится сравнение между абсолютным значением разности произведения переменных `guess` и `x`, и переменной `epsilon`. Для этого вычисляется выражение `guess*guess-x` и результат вычисления передается в функцию `absoluteValue`.

Значение, возвращаемое функцией `absoluteValue`, сравнивается со значением переменной `epsilon`. Если это значение больше или равно значению переменной `epsilon`, то это говорит о том, что требуемая точность еще не достигнута и вычисления должны продолжаться. При этом выполняется новая итерация и производится расчет следующего приблизительного значения.

Наконец, когда приблизительное значение будет достаточно близко к истинному значению, производится выход из цикла. После чего полученное значение возвращается в вызывающую программу. Внутри функции `main` возвращаемое значение передается в функцию `printf`, которая и отображает его на экране.

Вы можете заметить, что обе функции `absoluteValue` и `squareRoot` имеют формальный параметр с именем `x`. Но это не приводит к путанице и компилятор четко различает эти переменные.

В действительности каждая функция имеет свой собственный набор параметров. Поэтому формальный параметр `x`, используемый в функции `absoluteValue`, никак не связан с формальным параметром `x`, который применяется в функции `squareRoot`.

То же самое относится и к локальным переменным. Вы можете объявлять переменные с одними и теми же именами внутри каждой функции. Компилятор языка программирования C не будет путать эти переменные, поскольку локальные переменные могут использоваться только локально внутри каждой функции. Другими словами, область видимости локальных переменных ограничивается только той функцией, в которой они объявлены. Но, как вы узнаете из главы 11, "Указатели", в языке программирования C все же есть специальный механизм для доступа к локальным переменным их внешних функций.

Из проведенных выше рассуждений следует, что когда значение выражения `guess2-x` передается в функцию `absoluteValue`, то это значение присваивается формальному параметру `x` и это присваивание не оказывает никакого влияния на переменную `x` внутри функции `squareRoot`.

Объявление возвращаемых типов и типов параметров

Как уже отмечалось, компилятор по умолчанию предполагает, что функция возвращает значение типа `int`. Точнее, когда производится вызов функции, компилятор предполагает, что функция возвращает значение типа `int`, если не выполнены следующие требования.

1. Функция объявлена в программе до того, как произведен вызов функции.
2. Значение, возвращаемое функцией, объявлено до того, как произведен вызов функции.

В программе из листинга 8.8 функция `absoluteValue` описана до того, как компилятор обнаружил вызов этой функции в функции `squareRoot`. При этом еще до вызова функции `absoluteValue`, компилятор знает, что эта функция должна возвращать значение типа `float`. Но если бы функция `absoluteValue` была описана после функции `squareRoot`, то при вызове функции `absoluteValue` компилятор мог бы предположить, что эта функция возвращает целочисленное значение. Большинство компиляторов языка программирования C перехватывают эту ошибку и создают соответствующее диагностическое сообщение.

Для того чтобы использовать описание функции `absoluteValue` после описания функции `squareRoot` (или даже в другом файле, о чем говорится в главе 15), вы должны объявить тип возвращаемого значения до того, как функция `absoluteValue` будет вызвана. Такое объявление можно сделать или внутри функции `squareRoot`, или перед ней. Обычно такие объявления делаются в начале программы. Такие объявления используются не только для указания типа возвращаемого значения, но и для указания типов параметров, которые задаются в функции.

Для объявления функции `absoluteValue`, которая возвращает тип `float` и также принимает один аргумент типа `float`, используется следующее объявление.

```
float absoluteValue (float);
```

Как видно из этого объявления, здесь указан только тип возвращаемого значения и тип параметра, заключенный в круглые скобки. Можно дополнительно указать имя параметра после типа.

```
float absoluteValue (float x);
```

Это имя может отличаться от того, которое будет использовано при описании функции, но компилятор все равно его игнорирует.

Во избежание ошибок при объявлении функции, необходимо просто скопировать заголовок, который используется при описании функции, и перенести его в нужное место. Не забудьте поместить точку с запятой в конце объявления.

Если функция не принимает аргументов, используйте ключевое слово `void` в круглых скобках. Если функция не возвращает значений, то это тоже необходимо объявить, чтобы предупредить все попытки использования возвращаемого значения.

```
void calculateTriangularNumber (int n);
```

Если функция принимает переменное число аргументов (функции подобные `printf` или `scanf`), то компилятор также должен быть об этом информирован. Объявление

```
int printf (char *format, ...);
```

сообщает компилятору, что функция `printf` принимает указатель на символ в качестве первого аргумента (подробнее об этом будет сказано позже), а затем принимается некоторое число дополнительных аргументов (о чем говорит многоточие "..."). Функции `printf` и `scanf` объявлены в специальном файле `stdio.h`. Именно поэтому вы помещаете следующую строку в начало каждой программы.

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

Без этой строки компилятор может предположить, что функции `printf` и `scanf` принимают фиксированное число аргументов, и будет сгенерирован некорректный код.

Компилятор автоматически преобразовывает переданные аргументы в соответствующие типы при вызове функции, но только в том случае, если вы разместили описание функции или объявили функцию с нужными типами параметров в программе до вызова функции.

Ниже приведены несколько замечаний и предложений по использованию функций.

- Помните, по умолчанию компилятор предполагает, что любая функция возвращает значение типа `int`.
- Когда описываете функцию, которая возвращает значение типа `int`, описывайте ее явно.
- Когда описываете функцию, которая не возвращает значений, используйте ключевое слово `void`.
- Компилятор приводит аргументы к требуемым типам только в том случае, если до вызова функции было сделано объявление или описание функции.

Во избежание проблем, объявляйте каждую функцию вашей программы до первого вызова даже если они уже описаны (позже вы можете принять решение о переносе объявлений в другое место или даже в другой файл).

Проверка аргументов

Квадратный корень из отрицательного числа уводит вас из области реальных чисел в область мнимых чисел. Исходя из этого подумайте, что может произойти, если вы передадите отрицательное число в функцию `squareRoot`? При использовании метода Ньютона-Рафсона процесс вычисления квадратного корня никогда не закончится, поскольку приближительное значение никогда не будет приближаться к истинному значению в процессе проведения итераций. Поэтому требуемая точность никогда не будет достигнута и программа будет повторять итерации бесконечное число раз. Выполнение программы можно будет прекратить только с помощью специальных команд или нажатием комбинации клавиш `<Ctrl+C>`.

Очевидно, что программа должна быть модифицирована во избежание подобной ситуации при вызове функции. Вы должны будете ввести дополнительные проверки в вызывающую программу для того, чтобы не допустить непредвиденных ситуаций при вызове функции `squareRoot`. Хотя это решение по сути можно назвать правильным, но оно не является оптимальным. Может случиться, что при разработке некоторой программы вам понадобится использовать функцию `squareRoot`, но при этом вы вполне можете забыть, что она не может обрабатывать отрицательные числа, и в функцию будет передано отрицательное число. При этом программа будет выполняться бесконечно и вы не сразу сможете понять, что произошло.

Намного разумнее будет перенести решение проблемы в то место, где она возникает, и производить проверку аргументов в самой функции `squareRoot`. При этом необходимо производить проверку переменной `x` внутри функции, а затем можно отображать соответствующее сообщение при передаче отрицательного аргумента, после чего должен происходить немедленный возврат из функции без выполнения вычислений. О том, что функция не производила расчет квадратного корня, может свидетельствовать само возвращаемое значение, которое в этом случае должно быть равно `-1.0`.

Функция вычисления квадратного корня входит в состав стандартной библиотеки языка C и называется `sqrt`. Если передать отрицательное значение в эту функцию, то возникнет ошибка вывода за пределы допустимых значений. Значение, которое возвращается функцией в этом случае, зависит от реализации. В некоторых случаях, если отобразить это значение, то оно будет показано как `nan`, что означает “not a number” (нет значения).

Ниже приведена модифицированная функция `squareRoot`, в которой производится проверка аргументов, а также в программе приводится объявление прототипа функции `squareRoot`, о назначении которого мы уже говорили.

```
/* Функция расчета квадратного корня числа.
   Если в функцию передается отрицательное значение, то
   отображается сообщение и возвращается значение -1. */
float squareRoot (float x)
{
    const float epsilon = .00001;
    float guess = 1.0;
    float absoluteValue (float x);

    if ( x < 0 )
    {
        printf ("Negative argument to squareRoot.\n");
        return -1.0;
    }

    while ( absoluteValue (guess * guess - x) >= epsilon )
        guess = ( x / guess + guess ) / 2.0;

    return guess;
}
```

Если в качестве аргумента передается отрицательное число, то отображается соответствующее сообщение и в вызывающую программу немедленно возвращается значение `-1.0`. Если аргумент не является отрицательным, то производится вычисление квадратного корня, как описано выше.

Как можно видеть из модифицированной функции `squareRoot` (аналогичное решение использовано в последнем примере из главы 7, “Работа с массивами”), в функции используется более одного утверждения `return`. После того как выполняется утверждение `return`, происходит немедленный выход из функции и все утверждения, находящиеся после этого утверждения, пропускаются. Это свойство делает утверждение `return` идеальным для использования в том случае, когда функция не возвращает значения. Как уже отмечалось, в этом случае используется простейшая форма утверждения.

```
return;
```

где не указывается значение. Вполне очевидно, что если функция должна возвращать значение, эта форма утверждения не может быть использована.

Нисходящее программирование

Понимание того, что функции могут вызывать функции, которые в свою очередь вызывают функции и т.д., формирует основу для создания удобных, структурированных программ. В основной подпрограмме `main` из листинга 8.8 функция `squareRoot` вызывается несколько раз. Все детали, связанные с вычислением квадратного корня, заключены в функции `squareRoot` и никак не связаны с функцией `main`. Поэтому вы можете вызвать функцию еще до того, как вы опишете саму функцию: необходимо только сделать объявление функции с указанием параметров и возвращаемых значений.

Позже, когда мы продолжим писать код для функции `squareRoot`, мы будем использовать технику программирования сверху вниз. Мы будем вызывать функцию `absoluteValue`, не вникая в детали ее реализации. В этом случае очень важно понимать, что всегда можно разработать функцию, возвращающую абсолютное значение числа.

Такая техника программирования способствует более удобному и быстрому написанию программ и легкому их пониманию. Например, читая листинг 8.1, вы можете легко определить, что программа просто рассчитывает и отображает квадратные корни трех чисел. Нет необходимости вникать в детали, как на самом деле выполняется расчет квадратного корня. Но если вам необходимо вникнуть в детали, то вы всегда можете проанализировать код функции `squareRoot`. Аналогично, вы можете не вникать в детали реализации функции `absoluteValue`, но это не мешает пониманию работы функции `squareRoot`.

Для понимания работы функции `squareRoot` вовсе не обязательно знать все нюансы работы функции `absoluteValue`, но если в этом возникает необходимость, то всегда можно изучить код функции `absoluteValue`.

Функции и массивы

Наравне с обычными переменными, в качестве аргумента в функцию можно передать и элемент массива, а также весь массив. Для того чтобы передать отдельный элемент массива в функцию (что мы уже делали ранее в главе 7, когда использовали функцию `printf` для отображения элементов массива), его необходимо указать как аргумент функции, используя стандартный синтаксис для написания элементов массива. Поэтому для того, чтобы извлечь квадратный корень из элемента массива `averages[i]` и присвоить его переменной с именем `sq_root_result`, необходимо написать следующее.

```
sq_root_result = squareRoot (averages[i]);
```

В самой функции `squareRoot` ничего не надо изменять для того, чтобы в нее можно было передавать элементы массива. Точно так же, как и для простых переменных, значение элемента массива копируется в соответствующий формальный параметр при вызове функции.

Передача в функцию всего массива также не является сложной проблемой. Необходимо только указать имя массива без индексов при вызове функции. Для примера предположим, что переменная `gradeScores` объявлена как массив, содержащий 100 элементов. При этом выражение `minimum(gradeScores)` будет восприниматься

как передача всех 100 значений массива `gradeScores` в функцию `minimum`. Но с другой стороны, вполне понятно, что функция `minimum` должна быть готова к принятию всего массива, т.е. она должна быть описана с соответствующим типом формальных параметров. Описание функции `minimum` может быть выполнено подобно тому, как показано ниже.

```
int minimum (int values[100])
{
    ...
    return minValue;
}
```

В данном фрагменте описана функция `minimum`, которая возвращает значение типа `int` и принимает массив, содержащий 100 целочисленных элементов. Все ссылки формального параметра ссылаются на соответствующие элементы массива, который передается в функцию. И если, например, обратиться к элементу массива `values[4]`, то на самом деле это будет обращение к элементу массива `gradeScores[4]`.

Для первой программы, в которой демонстрируется функция, принимающая в качестве аргумента массив, будет написана функция `minimum`, которая находит минимальное значение в массиве из 10 элементов. Эта функция, вместе с подпрограммой `main`, в которой производится инициализация массива, показана в программе листинга 8.9.

Листинг 8.9. Определение минимального значения в массиве

```
// Function to find the minimum value in an array
#include <stdio.h>
int minimum (int values[10])
{
    int minValue, i;
    minValue = values[0];

    for ( i = 1; i < 10; ++i )
        if ( values[i] < minValue )
            minValue = values[i];

    return minValue;
}

int main (void)
{
    int scores[10], i, minScore;
    int minimum (int values[10]);

    printf ("Enter 10 scores\n");
    for ( i = 0; i < 10; ++i )
        scanf ("%i", &scores[i]);
    minScore = minimum (scores);
    printf ("\nMinimum score is %i\n", minScore);

    return 0;
}
```

Листинг 8.9. Вывод

```
Enter 10 scores
69
97
65
87
69
86
78
67
92
90
Minimum score is 65
```

Первое, что должно броситься в глаза, — это объявление прототипа функции `minimum` в подпрограмме `main`. При этом компилятор будет знать, что функция `minimum` возвращает целочисленное значение и принимает массив из 10 целых чисел. Хотя в данном случае и нет необходимости объявлять прототип функции, т.к. описание функции сделано до ее вызова в функции `main`, но всегда старайтесь объявлять все функции таким образом для того, чтобы получать надежный программный код.

После объявления массива `scores`, производится запрос к пользователю на ввод всех 10 значений. С помощью функции `scanf` производится размещение каждого значения массива в соответствующие ячейки массива `scores[i]`, где переменная `i` принимает значения от 0 до 9. После того как введены все значения, вызывается функция `minimum`, в которую массив передается в качестве аргумента.

Значения, связанные с именем формального параметра, используются для ссылок на элементы массива внутри функции. В соответствии с объявлением, это должен быть массив из 10 целых чисел. Локальная переменная `minValue` используется для хранения минимального значения массива и при инициализации ей присваивается значение первого элемента массива `values[0]`. Затем в цикле просматриваются остальные элементы массива и каждый элемент сравнивается со значением переменной `minValue`. Если значение очередного элемента массива `values[i]` меньше чем значение `minValue`, то это значение присваивается переменной `minValue` и становится текущим минимальным значением, и при этом продолжается проверка остальных элементов массива. В конечном итоге, переменной `minValue` будет присвоено самое минимальное значение из всего массива. После окончания цикла, значение переменной `minValue` возвращается в вызывающую программу, где оно присваивается переменной `minScore` и затем отображается на экране.

Исходя из поставленной задачи, в вышеприведенном случае можно было использовать только массив, содержащий 10 целых чисел. Поэтому вы можете использовать различные массивы, содержащие по 10 целочисленных элементов, и вызывать функцию `minimum` для каждого массива, если нужно найти минимальное значение. Аналогично этой функции, вы можете разработать функции для поиска максимального значения, медианы, среднего значения и т.д.

Создавая небольшие, независимые функции, которые хорошо выполняют отдельные, четко определенные задачи, вы можете в дальнейшем использовать их для решения более сложных задач и на их основе создавать соответствующие приложения. Например, вы можете последовательно разрабатывать функции для статистических расчетов, которые принимают массивы в качестве аргумента и последовательно выполняют расчеты медианы, стандартного отклонения и т.д. и на их основе создавать

программу статистических расчетов. Такая методология программирования позволяет разрабатывать программы, которые легко писать, понимать, модифицировать и устанавливать.

Разумеется, функция `minimum`, которая по определению должна быть универсальной функцией общего назначения, на самом деле не является таковой, т.к. она может работать только с массивами размером в 10 элементов. Но это легко исправить, вы можете повысить универсальность функции, которая будет принимать массивы любого размера. При объявлении функции вы должны пропустить спецификацию числа элементов, содержащуюся в объявлении формального параметра. На самом деле компилятор языка C игнорирует эту часть объявления формального параметра для массива. Все компиляторы воспринимают объявление массива, а не количество элементов в массиве.

В листинге 8.10 приведена улучшенная версия программы из листинга 8.9, в которой функция `minimum` находит наименьшее значение для массива любого размера.

Листинг 8.10. Улучшение функции для вычисления минимального значения в массиве

```
// Function to find the minimum value in an array
#include <stdio.h>
int minimum (int values[], int numberOfElements)
{
    int minValue, i;
    minValue = values[0];

    for ( i = 1; i < numberOfElements; ++i )
        if ( values[i] < minValue )
            minValue = values[i];

    return minValue;
}

int main (void)
{
    int array1[5] = { 157, -28, -37, 26, 10 };
    int array2[7] = { 12, 45, 1, 10, 5, 3, 22 };
    int minimum (int values[], int numberOfElements);

    printf ("array1 minimum: %i\n", minimum (array1, 5));
    printf ("array2 minimum: %i\n", minimum (array2, 7));

    return 0;
}
```

Листинг 8.10. Вывод

```
array1 minimum: -37
array2 minimum: 1
```

В этой программе функция `minimum` объявлена с двумя параметрами: первый параметр — это массив целых чисел, минимальное значение которого необходимо вычислить, а второй — это количество элементов массива. Между открывающей и закрывающей круглыми скобками содержится информация для компилятора, т.е. он узнает, что в функцию передаются массив целых чисел и целое число, определяющее количество

элементов массива. Как уже говорилось, компилятору не нужно знать истинный размер массива.

Формальный параметр `numberOfElements` используется вместо числа 10, которое ранее использовалось для задания числа элементов. Поэтому в цикле `for` производится просмотр значений элементов массива от второго элемента `values[1]` до последнего элемента, который будет равен `values[numberOfElements - 1]`. В подпрограмме `main` задаются два массива с 5 и 7 элементами соответственно.

При вызове функции `printf` первый раз производится расчет минимального значения первого массива с помощью функции `minimum`, в которую передаются аргументы `array1` и 5. Второй аргумент определяет количество элементов массива. Функция `minimum` определяет минимальное значение этого массива и возвращает результат -37, который и отображается на экране. Когда функция `minimum` вызывается во второй раз, в нее передается массив `array2` и количество элементов этого массива. Функцией возвращается значение 1, которое передается в функцию `printf` для отображения на экране.

Оператор присваивания

Познакомьтесь с программой из листинга 8.11 и постарайтесь предсказать ее вывод до того, как посмотрите на результат работы программы, приведенный в листинге.

Листинг 8.11. Изменение значений элементов массива

```
#include <stdio.h>
void multiplyBy2 (float array[], int n)
{
    int i;
    for ( i = 0; i < n; ++i )
        array[i] *= 2;
}

int main (void)
{
    float floatVals[4] = { 1.2f, -3.7f, 6.2f, 8.55f };
    int i;
    void multiplyBy2 (float array[], int n);

    multiplyBy2 (floatVals, 4);
    for ( i = 0; i < 4; ++i )
        printf ("%0.2f ", floatVals[i]);
    printf ("\n");

    return 0;
}
```

Листинг 8.11. Вывод

2.40 -7.40 12.40 17.10

Когда вы познакомитесь с программой из листинга 8.11, ваше внимание, несомненно, должно привлечь следующее утверждение.

```
array[i] *= 2;
```

Здесь встречается оператор, который мы еще не обсуждали, это составной оператор “умножить-равно”. Его действие заключается в том, что производится умножение левой части утверждения на правую, и результат сохраняется в левой части. Поэтому предыдущее утверждение полностью эквивалентно следующему утверждению.

```
array[i] = array[i] * 2;
```

Изучая программу, вы также должны обратить внимание, как функция `multiplyBy2` изменяет значения массива `floatVals`. Это не согласуется с тем, о чем мы говорили недавно: что функция не может изменять значения аргументов. Но это происходит.

В данной программе специально подчеркнут тот факт, о котором вы всегда должны помнить, — это особое поведение массивов. Если в функции производится изменение значений элементов переданного массива, то эти изменения будут произведены и в оригинальном массиве. Эти изменения останутся даже после того, как в функции будут произведены все вычисления и будет сделан возврат в вызывающую программу.

Причина, по которой поведение простых переменных или элементов массива, значения которых не изменяются в функции, отличаются от поведения всего массива, заслуживает дополнительного обсуждения. Как уже упоминалось, когда функция вызывается, то значения всех аргументов копируются в соответствующие формальные параметры. В этом случае это утверждение верно.

Но когда передается весь массив, то содержимое оригинального массива не копируется в формальный параметр типа массив. Вместо этого в функцию передается информация о местонахождении в памяти элементов массива. Поэтому все изменения, сделанные для формального параметра типа массив, на самом деле изменяют значения элементов оригинального массива, а не его копии, которой просто нет. Поэтому, когда происходит возврат из функции, все изменения, сделанные в массиве, остаются.

Помните, что все рассуждения об изменении значений элементов оригинального массива применимы только в том случае, когда в качестве аргумента передается весь массив, а не отдельные элементы массива, которые ведут себя аналогично простым переменным и, соответственно, их значения не могут изменяться в функции, а изменяются только значения их копий.

Сортировка элементов массива

Еще одним подтверждением того факта, что функция может изменять значения элементов оригинального массива, переданного в функцию, будет разработка функции сортировки (упорядочивания) элементов массива. Процесс сортировки всегда привлекает повышенное внимание программистов, поскольку операция сортировки является наиболее часто используемой. За небольшое время было разработано много замысловатых алгоритмов, среди которых и такие, которые используют минимально возможное количество оперативной памяти.

Поскольку целью этой книги не является обучение хитростям сортировки, то будет разработана функция, использующая простой алгоритм упорядочивания массива по возрастающим значениям. Сортировка массива по возрастающим значениям представляет элементы массива таким образом, что наименьшие значения будут находиться в начале массива, а наибольшие — в конце, т.е. значения элементов массива будут возрастать от начала массива к концу.

Если вы хотите сортировать массив из n элементов в возрастающем порядке, то вы можете сделать это с помощью последовательного сравнения элементов массива. Начиная с первого элемента и сравнивайте его со вторым элементом. Если значение первого элемента больше значения второго элемента, то вы должны просто поменять

их местами, т.е. переставить значения этих элементов. Затем сравните значение первого элемента массива (который на данном этапе является наименьшим) со значением третьего элемента. Аналогично предыдущему случаю, если значение первого элемента больше значения третьего элемента, то производится перестановка. В противном случае все остается без изменений. Таким образом, вы уже получили наименьшее значение для трех элементов массива.

Если повторять процесс до тех пор, пока не будут просмотрены все элементы массива, то по окончании просмотра в первом элементе массива будет находиться наименьшее значение всего массива.

Теперь аналогичный процесс можно повторить и со вторым элементом массива, сравнивая его с третьим, четвертым и т.д. элементами массива. После этого во втором элементе массива будет находиться наименьшее значение из значений элементов со второго по последний.

Думаю, что логика работы вам уже понятна — необходимо последовательно выполнить сравнение очередного элемента массива с оставшимися элементами и получить очередное наименьшее значение. По достижении предпоследнего элемента задача будет выполнена — массив будет отсортирован в возрастающем порядке.

Представленный ниже алгоритм дает формализованное описание описанного выше алгоритма сортировки. В этом алгоритме предполагается, что вы сортируете массив a из n элементов.

Простой алгоритм сортировки с помощью перестановок

Шаг 1. Установить i в 0.

Шаг 2. Установить j в $i+1$.

Шаг 3. Если $a[i] > a[j]$, переставить их значения.

Шаг 4. Установить j в $j+1$. Если $j < n$, перейти к шагу 3.

Шаг 5. Установить i в $i+1$. Если $i < n-1$, перейти к шагу 2.

Шаг 6. Массив a отсортирован в возрастающем порядке.

В программе из листинга 8.12 реализован представленный алгоритм в функции с именем `sort`, которая принимает два аргумента: массив для сортировки и число элементов массива.

Листинг 8.12. Сортировка массива целых чисел в возрастающем порядке

```
// Program to sort an array of integers into ascending order
#include <stdio.h>
void sort (int a[], int n)
{
    int i, j, temp;
    for ( i = 0; i < n - 1; ++i )
        for ( j = i + 1; j < n; ++j )
            if ( a[i] > a[j] ) {
                temp = a[i];
                a[i] = a[j];
                a[j] = temp;
            }
}

int main (void)
{
```

```

int i;
int array[16] = { 34, -5, 6, 0, 12, 100, 56, 22,
    44, -3, -9, 12, 17, 22, 6, 11 };
void sort (int a[], int n);

printf ("The array before the sort:\n");
for ( i = 0; i < 16; ++i )
    printf ("%i ", array[i]);
sort (array, 16);
printf ("\n\nThe array after the sort:\n");
for ( i = 0; i < 16; ++i )
    printf ("%i ", array[i]);
printf ("\n");

return 0;
}

```

Листинг 8.12. Вывод

```

The array before the sort:
34 -5 6 0 12 100 56 22 44 -3 -9 12 17 22 6 11
The array after the sort:
-9 -5 -3 0 6 6 11 12 12 17 22 22 34 44 56 100

```

В функции `sort` для реализации алгоритма использовано вложение циклов. Во внешнем цикле последовательно выбираются элементы массива от первого до предпоследнего (`a[n-2]`). Для каждого элемента, выбранного во внешнем цикле, запускается внутренний цикл, где со значением выбранного элемента сравниваются все последующие значения элементов массива.

Если нарушен возрастающий порядок, т.е. если `a[i]` больше чем `a[j]`, то значения элементов переставляются местами. Переменная `temp` используется для временного хранения значения при перестановке значений. Когда оба цикла закончатся, массив будет отсортирован в возрастающем порядке.

Выполнение функции на этом заканчивается. В подпрограмме `main` задается массив, который инициализируется 16-ю целочисленными значениями. Затем в программе производится отображение элементов массива и вызывается функция сортировки `sort`, в которую в качестве аргумента передается массив и число 16 — количество элементов массива. После возврата из функции, программа вновь отображает значения элементов массива. Как можно увидеть из вывода, сделанного программой, функция `sort` успешно отсортировала массив в возрастающем порядке.

Функция сортировки `sort`, используемая в программе, является довольно примитивной. Ценой этой простоты является скорость выполнения сортировки. Если с помощью такой функции сортировать очень большие массивы, содержащие сотни тысяч элементов, то у вас не хватит терпения дождаться конца сортировки. В таких случаях необходимо использовать более эффективные алгоритмы. Серия книг Дональда Кнута *Искусство программирования* является одним из наиболее известных учебников по таким алгоритмам.

В стандартной библиотеке языка C есть функция `qsort`, которая может использоваться для сортировки данных любых типов. Однако прежде чем использовать эту функцию, вы должны хорошо понимать назначение указателей на функции, которые обсуждаются в главе 11.

Многомерные массивы

Многомерные массивы могут быть переданы в функцию только как обычные переменные или элементы одномерного массива. Поэтому в утверждении

```
squareRoot (matrix[i][j]);
```

вызывается функция `squareRoot`, в которую в качестве аргумента передается значение, находящееся в матрице `matrix[i][j]`.

Полностью многомерный массив может быть передан в функцию как одномерный массив, для чего следует просто указать имя массива. Например, если матрица `measured_values` объявлена как двумерный массив целых чисел, то утверждение

```
scalarMultiply (measured_values, constant);
```

можно использовать для вызова функции, которая перемножает значение каждого элемента матрицы на заданную константу. Разумеется, при этом предполагается, что функция будет изменять значения оригинальной матрицы `measured_values`. Обсуждение, которое касалось одномерного массива, вполне применимо и в данном случае. Присваивания, которые производятся для элементов формального параметра типа массив внутри функции, изменяют и значения элементов оригинального массива, передаваемого в функцию.

Когда в качестве формального параметра объявляется одномерный массив, то вы уже знаете, что функции не нужно знать истинный размер массива. Поэтому используйте только пару квадратных скобок для того, чтобы информировать компилятор языка программирования C о том, что данный параметр является массивом. Но такой способ нельзя напрямую применить к многомерному массиву. Для двумерного массива можно пропустить количество строк, но в объявлении обязательно должно присутствовать количество столбцов массива. Поэтому объявления.

```
int array_values[100][50]
```

и

```
int array_values[][50]
```

будут правильными и в обоих случаях они свидетельствуют о массиве с именем `array_values`, содержащем 100 строк и 50 столбцов.

Но объявления

```
int array_values[100][]
```

или

```
int array_values[][]
```

не являются корректными, поскольку не указано количество столбцов массива.

В программе листинга 8.13 описана функция `scalarMultiply`, в которой производится умножение элементов двумерного целочисленного массива на целое число. В подпрограмме `main` функция `scalarMultiply` вызывается дважды. После каждого вызова массив передается в функцию `displayMatrix` для отображения содержимого массива. Обратите особое внимание на вложение циклов, которое используется как в функции `scalarMultiply`, так и в функции `displayMatrix` для последовательного доступа к каждому элементу двумерного массива.

Листинг 8.13. Использование многомерных массивов и функций

```
#include <stdio.h>
int main (void)
{
    void scalarMultiply (int matrix[3][5], int scalar);
    void displayMatrix (int matrix[3][5]);
    int sampleMatrix[3][5] =
        {
            { 7, 16, 55, 13, 12 },
            { 12, 10, 52, 0, 7 },
            { -2, 1, 2, 4, 9 }
        };
    printf ("Original matrix:\n");
    displayMatrix (sampleMatrix);
    scalarMultiply (sampleMatrix, 2);
    printf ("\nMultiplied by 2:\n");
    displayMatrix (sampleMatrix);
    scalarMultiply (sampleMatrix, -1);
    printf ("\nThen multiplied by -1:\n");
    displayMatrix (sampleMatrix);

    return 0;
}

// Function to multiply a 3 x 5 array by a scalar
void scalarMultiply (int matrix[3][5], int scalar)
{
    int row, column;

    for ( row = 0; row < 3; ++row )
        for ( column = 0; column < 5; ++column )
            matrix[row][column] *= scalar;
}

void displayMatrix (int matrix[3][5])
{
    int row, column;

    for ( row = 0; row < 3; ++row ) {
        for ( column = 0; column < 5; ++column )
            printf ("%5i", matrix[row][column]);
        printf ("\n");
    }
}
```

Листинг 8.13. Вывод

```
Original matrix:
7 16 55 13 12
12 10 52 0 7
-2 1 2 4 9
Multiplied by 2:
14 32 110 26 24
24 20 104 0 14
```

```
-4 2 4 8 18
Then multiplied by -1:
-14 -32 -110 -26 -24
-24 -20 -104 0 -14
4 -2 -4 -8 -18
```

В подпрограмме `main` сначала описывается матрица `sampleValues`, которая затем передается в функцию `displayMatrix` для отображения начальных значений на экране. Внутри функции `displayMatrix` используется вложение циклов. Во внешнем цикле производится последовательный перебор всех строк матрицы, поэтому значения переменной `row` изменяются от 0 до 2. Для каждого значения переменной `row` выполняется внутренний цикл. В этом цикле производится перебор всех столбцов для каждой строки, поэтому переменная `column` изменяется от 0 до 4.

В утверждении `printf` производится отображение значения, содержащегося на пересечении заданных строки и столбца, используется символ форматирования `%5i`, чтобы отображаемые на экране значения были выровнены. По завершении работы внутреннего цикла отображаются значения заданной строки, после чего используется переход на новую строку и следующая строка матрицы будет отображаться на очередной строке экрана.

При первом вызове функции `scalarMultiply` задается, что элементы массива `sampleMatrix` будут умножены на число 2. Внутри функции используется последовательность вложенных циклов для доступа к каждому элементу массива. При этом элемент матрицы `matrix[row][column]` умножается на значение переменной `scalar` с помощью оператора умножения `"*="`. После выполнения функции и возврата в подпрограмму `main`, еще раз вызывается функция `displayMatrix` для отображения содержимого элементов матрицы. Исходя из вывода, сделанного программой, становится очевидным, что элементы матрицы в самом деле были умножены на 2.

Функция `scalarMultiply` во второй раз вызывается для того, чтобы умножить уже измененные элементы матрицы на -1. Модифицированный массив затем отображается в последний раз путем вызова функции `displayMatrix`, и на этом выполнение программы завершается.

Многомерный массив переменной длины и функции

Вы можете использовать преимущества массивов переменной длины, которые разрешены в языке программирования C, и написать функцию, которая будет принимать многомерные массивы переменных размеров. Например, программу из листинга 8.13 можно переписать так, что функции `scalarMultiply` и `displayMatrix` будут принимать массивы, содержащие любое количество строк и столбцов. Такая программа показана в листинге 8.13А.

Листинг 8.13А. Многомерный массив переменной длины

```
#include <stdio.h>
int main (void)
{
    void scalarMultiply (int nRows, int nCols,
        int matrix[nRows][nCols], int scalar);
    void displayMatrix (int nRows, int nCols, int matrix[nRows][nCols]);
    int sampleMatrix[3][5] =
    {
        { 7, 16, 55, 13, 12 },
```

```

        { 12, 10, 52, 0, 7 },
        { -2, 1, 2, 4, 9 }
    };

    printf ("Original matrix:\n");
    displayMatrix (3, 5, sampleMatrix);
    scalarMultiply (3, 5, sampleMatrix, 2);
    printf ("\nMultiplied by 2:\n");
    displayMatrix (3, 5, sampleMatrix);
    scalarMultiply (3, 5, sampleMatrix, -1);
    printf ("\nThen multiplied by -1:\n");
    displayMatrix (3, 5, sampleMatrix);

    return 0;
}

// Функция умножения матрицы на скаляр.
void scalarMultiply (int nRows, int nCols,
                    int matrix[nRows][nCols], int scalar)
{
    int row, column;

    for ( row = 0; row < nRows; ++row )
        for ( column = 0; column < nCols; ++column )
            matrix[row][column] *= scalar;
}

void displayMatrix (int nRows, int nCols, int matrix[nRows][nCols])
{
    int row, column;

    for ( row = 0; row < nRows; ++row ) {
        for ( column = 0; column < nCols; ++column )
            printf ("%5i", matrix[row][column]);
        printf ("\n");
    }
}

```

Листинг 8.13А. Вывод

```

Original matrix:
7 16 55 13 12
12 10 52 0 7
-2 1 2 4 9
Multiplied by 2:
14 32 110 26 24
24 20 104 0 14
-4 2 4 8 18
Then multiplied by -1:
-14 -32 -110 -26 -24
-24 -20 -104 0 -14
4 -2 -4 -8 -18

```

Объявление функции scalarMultiply выглядит следующим образом.

```
void scalarMultiply (int nRows, int nCols, int matrix[nRows][nCols],
int scalar)
```

Строки и колонки матрицы, `nRows` и `nCols`, должны быть помещены в качестве параметров до того, как будет записана сама матрица, чтобы компилятор знал об этих параметрах и мог их учесть. Если объявление записать следующим образом.

```
void scalarMultiply (int matrix[nRows][nCols], int nRows, int nCols,
int scalar)
```

то будет сгенерирована ошибка компиляции, поскольку компилятор ничего не будет знать о переменных `nRows` и `nCols` до того, как он встретит объявление матрицы.

Как и следовало ожидать, вывод, сделанный программой из листинга 8.13А, полностью соответствует выводу, сделанному программой из листинга 8.13. Благодаря использованию массивов переменной длины теперь у вас есть две функции (`scalarMultiply` и `displayMatrix`), которые вы можете использовать при работе с любыми матрицами.

Как уже отмечалось, вы должны убедиться, что ваш компилятор языка программирования C полностью поддерживает массивы переменной длины, чтобы использовать эти возможности.

Глобальные переменные

Итак, пришло время обобщить положения, с которыми вы ознакомились в этой главе, а также изучить новые правила языка C. Возьмем программу из листинга 7.7, в которой положительное число преобразовывается в число с другим основанием счисления и это оформлено в виде функции. Для этого вы должны концептуально разделить программу на логические сегменты. Вполне очевидно, что такое разделение сделано с помощью комментариев внутри подпрограммы `main`. Они предваряют три основные задачи, которые выполняются в программе: прием числа и основания системы счисления от пользователя, преобразование числа в заданную систему счисления и отображение результата.

Поэтому вы можете создать три функции, выполняющие аналогичные задачи. Первая функция, которая будет вызвана, — это функция `getNumberAndBase`. С ее помощью будет сделан запрос к пользователю на ввод необходимого числа и основания системы счисления для преобразования и затем эти значения будут считаны с экрана терминала. При этом стоит сделать небольшое улучшение кода, приведенного в листинге 7.7. То есть если пользователь введет значение основания системы счисления меньше чем 2 или больше чем 16, то программа должна отобразить соответствующее сообщение на экране и установить для основания системы счисления значение 10. При этом после завершения программы повторяется отображение значений, введенных пользователем. Можно изменить поведение программы и заставить ее повторять запрос на ввод значений для получения допустимых величин, но это мы оставляем читателю в качестве упражнения.

Вторую функцию, которую необходимо вызвать, можно назвать `convertNumber`. Назначение этой функции заключается в приеме значения, которое ввел пользователь, и преобразовании его к другой системе счисления. При этом результат сохраняется в массиве `convertedNumber`.

Третья и последняя функция должна отображать результат и ее можно назвать `displayConvertedNumber`. Эта функция принимает значения, сохраненные в массиве `convertedNumber`, и отображает их на экране в правильном порядке. Для каждой отображаемой цифры выполняется преобразование в соответствии с системой

счисления, при этом массив `baseDigits` используется таким образом, что отображается необходимый символ.

Все три функции, которые мы только что определили, связаны между собой с помощью *глобальных переменных*. Как уже отмечалось выше, одним из фундаментальных свойств локальных переменных является то, что их значения могут быть доступны только в пределах функции, в которой эти переменные объявлены. Как вы уже, возможно, догадались, ограничение не применимо к глобальным переменным. Поэтому значение глобальной переменной может быть доступно для любой функции программы.

Отличительной особенностью объявления глобальной переменной является то, что оно должно быть выполнено за пределами всех функций, а не как объявление локальных переменных в какой-либо функции. Поэтому глобальная переменная не принадлежит ни одной из функций. Все функции программы имеют доступ к глобальным переменным и при необходимости могут изменять их значения.

В программе из листинга 8.14 объявлены четыре глобальные переменные. Каждая из них используется по крайней мере двумя функциями программы. Поскольку массив `baseDigits` и переменная `nextDigit` используются только в функции `displayConvertedNumber`, они объявлены не как глобальные, а как локальные переменные, и их значения используются локально в функции `displayConvertedNumber`.

Глобальные переменные объявляются в начале программы. Поскольку они не находятся в пределах какой-либо функции, то к ним можно обращаться из любой функции программы.

Листинг 8.14. Преобразование числа в другую систему счисления

```
// Программа преобразовывает положительное целое число в другую
// систему счисления.
#include <stdio.h>
int convertedNumber[64];
long int numberToConvert;
int base;
int digit = 0;
void getNumberAndBase (void)
{
    printf ("Number to be converted? ");
    scanf ("%li", &numberToConvert);
    printf ("Base? ");
    scanf ("%i", &base);
    if ( base < 2 || base > 16 ) {
        printf ("Bad base - must be between 2 and 16\n");
        base = 10;
    }
}

void convertNumber (void)
{
    do {
        convertedNumber[digit] = numberToConvert % base;
        ++digit;
        numberToConvert /= base;
    }
    while ( numberToConvert != 0 );
}

void displayConvertedNumber (void)
```

```

{
    const char baseDigits[16] =
        { '0', '1', '2', '3', '4', '5', '6', '7',
          '8', '9', 'A', 'B', 'C', 'D', 'E', 'F' };
    int nextDigit;

    printf ("Converted number = ");
    for (--digit; digit >= 0; --digit ) {
        nextDigit = convertedNumber[digit];
        printf ("%c", baseDigits[nextDigit]);
    }
    printf ("\n");
}

int main (void)
{
    void getNumberAndBase (void), convertNumber (void),
      displayConvertedNumber (void);
    getNumberAndBase ();
    convertNumber ();
    displayConvertedNumber ();

    return 0;
}

```

Листинг 8.14. Вывод

```

Number to be converted? 100
Base? 8
Converted number = 144

```

Листинг 8.14. Вывод (Повторение)

```

Number to be converted? 1983
Base? 0
Bad base - must be between 2 and 16
Converted number = 1983

```

Обратите внимание, как разумный выбор имен функций в листинге 8.14 упрощает понимание логики работы программы. Можно читать последовательно подпрограмму `main` и узнавать о выполняемых действиях: получить число и основание системы счисления, преобразовать число и наконец отобразить преобразованное число. Такое разбиение программы уже описывалось в главе 7 и является прямым следствием структурированного стиля программирования, при котором программа разбивается на небольшие, четко выраженные задачи. При этом отпадает необходимость в расстановке комментариев, которые обычно ставятся перед вызовом функцией, т.к. назначение функции становится ясным из ее названия.

Основным назначением глобальных переменных в функции является обеспечение удобного доступа к одним и тем же переменным из многих функций. Вместо того чтобы передавать необходимые значения в каждую функцию с помощью аргументов, функция просто обращается к нужной глобальной переменной. Но такое удобство

имеет свои издержки. Поскольку функция явно ссылается на глобальную переменную, использование функции будет несколько ограниченным, поскольку необходимо каждый раз проверять наличие в программе глобальной переменной с заданным именем.

Например, функция `convertNumber` из листинга 8.14 успешно преобразовывает только числа, которые сохранены в глобальной переменной с именем `numberToConvert`, в число с другим основанием системы счисления, значение которого сохранено в переменной `base`. К тому же должны быть инициализированы переменная `digit` и массив `convertedNumber`. Поэтому более гибкой и удобной версией этой функции будет функция, в которую можно передать все необходимые аргументы.

Хотя использование глобальных переменных позволяет сократить число аргументов, которые необходимо передавать в функции, последствием такого решения может быть потеря универсальности функций и, в некоторый случаях, ухудшение читабельности программы. Ухудшение читабельности программы обусловлено тем, что использование глобальных переменных снижает количество параметров, используемых при объявлении функции, а это приводит к более трудному пониманию назначения функции. Также при просмотре вызова отдельной функции для пользователя не предоставляется необходимая информация о типах параметров, которые необходимы функции для нормальной работы.

Некоторые программисты используют соглашение об использовании префиксов для всех глобальных переменных и в качестве такого префикса рекомендуют букву "g". Например, переменные, объявленные в листинге 8.14, должны выглядеть следующим образом.

```
int gConvertedNumber[64];
long int gNumberToConvert;
int gBase;
int gDigit = 0;
```

Удобство заключается в том, что становится легче различать глобальные и локальные переменные и, соответственно, легче читать программу. Например, при чтении утверждения.

```
nextMove = gCurrentMove + 1;
```

становится ясно, что `nextMove` является локальной переменной, а `gCurrentMove` представляет собой глобальную переменную. Для пользователя это дополнительная информация об области видимости этих переменных, благодаря чему он легче найдет исходные объявления данных переменных.

Приведем еще одно замечание о глобальных переменных. По умолчанию они должны быть инициализированы нулем. Поэтому при объявлении глобальной переменной

```
int gData[100];
```

все 100 элементов массива `gData` в начале выполнения программы будут установлены в нуль. Но, в отличие от глобальных переменных, локальные переменные по умолчанию не устанавливаются в нуль и должны быть явно инициализированы в программе.

Автоматические и статические переменные

Когда вы объявляете локальные переменные внутри функции обычным способом, подобно объявлению переменных `guess` и `epsilon` в функции `squareRoot`:

```
float squareRoot (float x)
{
    const float epsilon = .00001;
    float guess = 1.0;
    . . .
}
```

то вы объявляете *автоматические локальные переменные*. Вспомните, что объявлению таких переменных может предшествовать ключевое слово `auto`, которое в данном случае является избыточным, т.к. по умолчанию все переменные, объявляемые в функциях и не имеющие дополнительных опций, являются автоматическими. Такие переменные автоматически создаются каждый раз при вызове функции.

В предыдущем примере локальные переменные `epsilon` и `guess` будут создаваться каждый раз, когда будет вызываться функция `squareRoot`. Как только функция `squareRoot` закончит работу, эти локальные переменные “исчезнут”. Этот процесс происходит автоматически, поэтому данные переменные и называются автоматическими переменными.

Автоматическим локальным переменным можно присваивать начальные значения, как это сделано для переменных `epsilon` и `guess`. Значения автоматических переменных рассчитываются и присваиваются им каждый раз при вызове функции. (Для констант, переменных с ключевым словом `const`, значения сохраняются в специальной памяти для чтения и не инициализируются при вызове функции.)

Но если вы поставите в начале объявления переменной ключевое слово `static`, то в силу вступают уже другие правила. Ключевое слово `static` в языке программирования C никак не связано с электрическими зарядами, а только отмечает тот факт, что нечто является постоянным. Это является ключевой концепцией статических переменных — они не создаются и не исчезают при вызове и окончании работы функции. Значения таких переменных остаются такими, какими они были при последнем вызове функции, и не изменяются до следующего вызова.

Статические локальные переменные отличаются и способом инициализации. Эти переменные инициализируются один раз в начале выполнения программы, а не каждый раз при вызове функции, в которой они объявлены. К тому же инициализирующее выражение должно быть простой константой или константным выражением. По умолчанию статические переменные инициализируются нулем, в отличие от автоматических переменных, которым не присваивается значение по умолчанию.

В функции `auto_static`, которая объявлена следующим образом

```
void auto_static (void)
{
    static int staticVar = 100;
    . . .
}
```

переменная `staticVar` инициализируется значением 100 только один раз в начале выполнения программы.

Для того чтобы присваивать статической переменной значение 100 при каждом вызове функций, необходимо использовать явное присваивание, как это показано ниже.

```
void auto_static (void)
{
    static int staticVar;
    staticVar = 100;
    . . .
}
```

Разумеется, такое использование переменной `staticVar` зависит от поставленной задачи и подобно применению автоматических переменных.

Программа из листинга 8.15 поможет вам лучше понять концепцию автоматических и статических переменных.

Листинг 8.15. Демонстрация применения статических и автоматических переменных

```
// Program to illustrate static and automatic variables
#include <stdio.h>
void auto_static (void)
{
    int autoVar = 1;
    static int staticVar = 1;

    printf ("automatic = %i, static = %i\n", autoVar, staticVar);
    ++autoVar;
    ++staticVar;
}

int main (void)
{
    int i;
    void auto_static (void);

    for ( i = 0; i < 5; ++i )
        auto_static ();

    return 0;
}
```

Листинг 8.15. Вывод

```
automatic = 1, static = 1
automatic = 1, static = 2
automatic = 1, static = 3
automatic = 1, static = 4
automatic = 1, static = 5
```

Внутри функции `auto_static` объявлены две локальные переменные. Первая переменная с именем `autoVar` является автоматической переменной типа `int`, которая инициализирована значением 1. Вторая переменная называется `staticVar` и является статической переменной. Она также имеет тип `int` и также инициализирована значением 1. В функции `auto_static` вызывается функция `printf` для отображения значений этих двух переменных. После этого переменные инкрементируются и выполнение функции заканчивается.

В подпрограмме `main` пять раз выполняется вызов функции `auto_static`. Благодаря выводу программы из листинга 8.15, становится очевидным различие между значениями этих двух переменных. Значение автоматической переменной остается равным 1 на каждой строке вывода. Это происходит потому, что каждый раз при вызове функции она инициализируется и ей присваивается значение 1. С другой стороны, вполне очевидно, что выводимые значения для статической переменной увеличиваются на 1: от 1 до 5. Это происходит потому, что статическая переменная инициализируется только один раз при запуске программы и ее значение возрастает при каждом вызове функции `auto_static`.

Выбор переменных (статических или автоматических) зависит от поставленной задачи. Если вы хотите, чтобы переменные сохраняли свое значение от одного вызова функции до другого (например, если вы подсчитываете, сколько раз вызывалась функция), то используйте статическую переменную. Или если необходимо, чтобы переменная инициализировалась только один раз и больше не изменялась, то также надо использовать статические переменные, т.к. значение такой переменной не будет изменяться при неоднократных вызовах функции. Особенно важно это использовать при работе с массивами.

С другой стороны, если необходимо устанавливать значение переменной при каждом вызове функции, то следует использовать автоматические переменные.

Рекурсивные функции

В языке программирования C поддерживается такая техника, как *рекурсия* функций. Рекурсивные функции используются для быстрого и эффективного решения проблем. Особенно в приложениях, где решение задачи может быть выражено с помощью последовательного использования одного и того же алгоритма. Одним из примеров может быть вычисление выражения, содержащего множество вложенных выражений, заключенных в круглые скобки. Наиболее распространенные из таких приложений используют поиск и сортировку структур данных, называемых *деревьями* и *списками*.

Хорошей иллюстрацией для использования рекурсивных функций является расчет факториала числа. Как известно, факториалом целого положительного числа n является произведение всех последовательных целых чисел от 1 до n , что обозначается $n!$. Факториал нуля является специальным случаем и ему присваивается значение 1. Итак, факториалы чисел 5 и 6 можно рассчитать следующим образом.

$$5! = 5 * 4 * 3 * 2 * 1 = 120$$

и

$$6! = 6 * 5 * 4 * 3 * 2 * 1 = 720$$

Сравнивая значения факториалов для чисел 5 и 6, можно заметить, что первое значение в 6 раз меньше второго. Поэтому можно записать: $6! = 6 * 5!$. В общем случае факториал любого положительного целого числа n , большего нуля, равен числу n , умноженному на факториал предыдущего числа $(n-1)$.

$$n! = n * (n - 1)!$$

Выражение значения $n!$ в терминах значения $(n-1)!$ называется рекурсивным определением, поскольку значение одного факториала основано на значении другого факториала. Поэтому вы легко можете разработать функцию, которая рассчитывает значение факториала положительного целого числа n в соответствии с рекурсивным определением. Такая функция используется в программе листинга 8.16.

Листинг 8.16. Рекурсивный расчет факториала

```
#include <stdio.h>
int main (void)
{
    unsigned int j;
    unsigned long int factorial (unsigned int n);
    for ( j = 0; j < 11; ++j )
        printf ("%2u! = %lu\n", j, factorial (j));

    return 0;
}

// Рекурсивная функция для расчета факториала положительного целого
числа.
unsigned long int factorial (unsigned int n)
{
    unsigned long int result;
    if ( n == 0 )
        result = 1;
    else
        result = n * factorial (n - 1);
    return result;
}
```

Листинг 8.16. Вывод

```
0! = 1
1! = 1
2! = 2
3! = 6
4! = 24
5! = 120
6! = 720
7! = 5040
8! = 40320
9! = 362880
10! = 3628800
```

То, что в функции `factorial` используется вызов самой функции, делает ее рекурсивной. Когда функция вызывается для расчета факториала числа 3, значением формального параметра `n` становится число 3. Поскольку это значение не равно нулю, то выполняется утверждение

```
result = n * factorial (n - 1);
```

которое возвращает результат в соответствии со следующим утверждением.

```
result = 3 * factorial (2);
```

В этом утверждении вновь вызывается функция `factorial` с целью рассчитать значение факториала числа 2. Поэтому расчет выражения для умножения числа 3 на факториал числа 2 задерживается на время выполнения расчета факториала числа 2.

Даже тогда, когда вы вновь вызываете одну и ту же функцию, вы должны помнить, что это вызов отдельной функции. Каждый раз при вызове функции в языке

программирования C, рекурсивной или нет, она получает набор своих собственных локальных переменных и формальных параметров, с которыми она будет работать. Поэтому локальная переменная `result` и формальный параметр `n`, что уже существуют в функции `factorial`, которая производит расчет факториала для числа 3, будут отличаться от переменной `result` и параметра `n`, которые будут созданы для расчета факториала числа 2.

Когда значение параметра `n` равно 2, функция факториал выполняется в утверждении

```
result = n * factorial (n - 1);
```

которое эквивалентно следующему утверждению.

```
result = 2 * factorial (1);
```

Еще раз напомним, что при умножении числа 2 на функцию вычисления факториала числа 1, происходит задержка на время вычисления факториала числа 1.

Когда значение переменной `n` станет равным 1, функция выполняется один раз в утверждении

```
result = n * factorial (n - 1);
```

которое эквивалентно следующему утверждению.

```
result = 1 * factorial (0);
```

Когда функция факториал вызывается для расчета факториала числа 0, значение переменной `result` устанавливается равным 1 и происходит возврат из функции, что приводит к последовательному вычислению всех задержанных выражений. Так как значение факториала числа 0 равно 1, то это значение возвращается в вызывающую функцию, которая также является функцией вычисления факториала, умножается на 1 и присваивается переменной `result`.

Число 1, которое является значением факториала числа 1, затем передается в вызывающую функцию, которая вновь становится функцией вычисления факториала, умножается на 2, сохраняется в переменной `result`, и возвращается значение факториала числа 2. Наконец, последнее возвращаемое значение 2 умножается на число 3 и вычисление факториала числа 3 завершается. Результирующее значение 6 возвращается как окончательный результат вычисления функции `factorial` и отображается с помощью функции `printf`.

Если суммировать последовательность операций, которые выполняются при расчете факториала числа 3, то получим следующую последовательность выражений.

```
factorial (3) = 3 * factorial (2)
= 3 * 2 * factorial (1)
= 3 * 2 * 1 * factorial (0)
= 3 * 2 * 1 * 1
= 6
```

Рекомендуем взять карандаш и бумагу и последовательно расписать все операции при вычислении факториала числа. Предположите, что вызывается функция для расчета факториала числа 4. Запишите значения переменных `n` и `result` для каждого вызова функции `factorial`.

Завершая обсуждение функций и переменных, не лишним будет повторить, что функции в языке программирования C являются очень мощным инструментом. К тому же они считаются ключевым звеном при структурировании программ.

Функции будут использоваться нами на протяжении всей оставшейся части книги.

Упражнения

1. Наберите и выполните все 16 программ, представленных в данной главе. Сравните вывод, сделанный вашей программой, с выводом, приведенным в книге для каждой программы.
2. Модифицируйте программу из листинга 8.4 таким образом, чтобы значение переменной `triangularNumber` возвращалось функцией. Затем вернитесь к программе из листинга 5.5 и измените ее так, чтобы в ней производился вызов новой версии функции `calculateTriangularNumber`.
3. Измените программу из листинга 8.8 так, чтобы значение переменной `epsilon` передавалось в функцию как аргумент. Поэкспериментируйте с различными значениями переменной `epsilon` и проверьте правильность получаемых результатов.
4. Измените программу из листинга 8.8 так, чтобы значение переменной `guess` выводилось на экран каждый раз, когда выполняется цикл `while`. Обратите внимание, как быстро значение переменной `guess` приближается к истинному квадратному корню. Какие выводы вы можете сделать, анализируя количество итераций при расчете квадратного корня в зависимости от начального значения переменной `guess`?
5. Критерии, используемые для выхода из цикла в функции `squareRoot` листинга 8.8, не подходят при вычислении квадратных корней из очень больших или очень маленьких чисел. Прежде чем сравнивать значения переменных `x` и `guess`², в программе должно быть произведено сравнение отношений этих двух значений с единицей. Чем ближе это отношение к единице, тем точнее результат приближается к истинному значению квадратного корня. Модифицируйте программу из листинга 8.8 с целью введения новых критериев.
6. Модифицируйте программу из листинга 8.8 таким образом, чтобы функция `squareRoot` принимала аргументы двойной точности и возвращала значения двойной точности. Не забудьте изменить значение переменной `epsilon`, которое должно учитывать использование переменных двойной точности.
7. Напишите функцию, которая возводит целое число в положительную степень. Назовите функцию `x_to_the_n` и используйте при вызове функции два аргумента `x` и `n`. Функция должна возвращать значение типа `long int`, которое представляет результат вычисления x^n .
8. Равенство, записанное в виде $ax^2+bx+c=0$, называется квадратным уравнением. Значения коэффициентов `a`, `b` и `c` для предыдущего примера являются константами. Поэтому равенство $4x^2-17x-15=0$ представляет квадратное уравнение, в котором `a=4`, `b=-17` и `c=-15`. Значения переменной `x`, которые удовлетворяют данному квадратному уравнению, называются корнями уравнения и могут быть рассчитаны с учетом значений `a`, `b` и `c` по следующим двум формулам.

Если значение выражения b^2-4ac , называемого дискриминантом, меньше нуля, корни квадратного уравнения являются мнимыми.

Напишите программу, которая находит корни квадратного уравнения. Программа должна позволять пользователю вводить значения коэффициентов `a`, `b` и `c`. Если дискриминант меньше нуля, должно отображаться сообщение о том, что значениями для корней являются мнимые числа. (Подсказка: используйте функцию `squareRoot`, которую вы разработали в этой главе.)

9. Общим наименьшим кратным (least common multiple – lcm) для двух положительных чисел u и v является наименьшее целое число, которое делится без остатка на оба числа. Таким образом, общим наименьшим кратным для чисел 15 и 10 (что записывается как $\text{lcm}(15, 10)$) будет число 30, поскольку 30 является наименьшим целым числом, которое без остатка делится как на 15, так и на 10. Напишите функцию `lcm`, которая принимает два целых числа и возвращает их общее наименьшее кратное. Эта функция должна производить расчет с использованием функции `gcd` из листинга 8.6 в соответствии со следующим тождеством:

$$\text{lcm}(u, v) = uv / \text{gcd}(u, v) \quad u, v \geq 0$$

10. Напишите функцию `prime`, которая возвращает 1, если аргумент является простым числом, и 0 – в противном случае.
11. Напишите функцию с именем `arraySum`, которая принимает два аргумента: массив целых чисел и число элементов массива. Функция должна возвращать сумму элементов массива.
12. Матрица M с i строками и j столбцами может быть транспонирована в матрицу N , имеющую j строк и i столбцов, путем установки значения $N_{a,b}$ равным значению $M_{b,a}$ для всех значений a и b .
- а) Напишите функцию `transposeMatrix`, которая принимает в качестве аргументов матрицы размером 4×5 и 5×4 . Функция должна транспонировать матрицу 4×5 в матрицу 5×4 . Также напишите программу для проверки этой функции.
 - б) Используйте массивы переменной длины и измените функцию `transposeMatrix` таким образом, чтобы она дополнительно принимала в качестве аргументов количество строк и столбцов и транспонировала матрицу.
13. Модифицируйте программу `sort` из листинга 8.12 таким образом, чтобы она принимала третий аргумент, указывающий, в каком порядке должен быть отсортирован массив. Затем модифицируйте алгоритм таким образом, чтобы в нем учитывался этот аргумент.
14. Перепишите все функции, разработанные в последних четырех упражнениях, таким образом, чтобы они использовали глобальные переменные вместо аргументов. Например, в предыдущем упражнении теперь должен сортироваться глобально объявленный массив.
15. Модифицируйте программу из листинга 8.14 таким образом, чтобы программа вновь запрашивала пользователя на ввод значения для основания системы счисления, если введено неправильное значение.
16. Модифицируйте программу из листинга 8.14 таким образом, чтобы пользователь мог преобразовывать любое количество целых чисел. Выход из программы должен производиться тогда, когда в качестве числа для преобразования будет введен 0.

Структуры

В главе 7, “Массивы”, представлены массивы, которые позволяют группировать элементы одного типа в единый логический модуль. Для обращения к элементу массива все, что необходимо сделать, — это написать имя массива с соответствующим индексом. В языке программирования С есть и другие конструкции для группировки элементов. Для таких конструкций используется общее название *структура*, и именно структуры будут предметом рассмотрения настоящей главы. Как вы увидите, структуры являются удобной и мощной конструкцией, которую вы обязательно будете использовать в своих разрабатываемых программах.

Предположим, что вы хотите в своей программе, сохранить дату, например 9/25/04, для ее использования при выводе данных или при некоторых расчетах.

Наиболее приемлемым способом хранения даты, принимая во внимание имеющийся уровень знаний, будет простое присваивание номера месяца целочисленной переменной с именем `month`, номера дня целочисленной переменной `day` и года целочисленной переменной `year`. Поэтому утверждения

```
int month = 9, day = 25, year = 2004;
```

будут хорошо справляться с поставленной задачей. Но это не всегда самое удачное решение. Предположим, в программе необходимо хранить данные о покупках. При этом вы должны будете создать три процедуры, `purchaseMonth`, `purchaseDay` и `purchaseYear`, которые вы должны будете вызывать при необходимости ввести дату.

В данном случае вы будете использовать три отдельные переменные для каждой даты, которую вы будете хранить в программе. Но учитывая, что эти переменные логически взаимосвязаны, гораздо целесообразнее объединить их в группу, содержащую множество таких данных, что достаточно легко сделать, используя структуры.

Структура для хранения даты

На языке программирования С можно описать структуру с именем `date`, которая содержит три элемента, представляющие месяц (`month`), день (`day`) и год (`year`). Синтаксис такой структуры вполне понятен и выглядит он следующим образом.

```
struct date
{
    int month;
    int day;
```



```
    int year;
};
```

Как видим, структура `date`, имеет три члена с именами `month`, `day` и `year`. Вполне логично, что описание структуры `date` является объявлением нового типа в языке, а значит, любой переменной можно присвоить этот тип, т.е. использовать сочетание `struct date` и объявить переменную следующим образом.

```
struct date today;
```

Аналогично можно объявить переменную `purchaseDate` того же самого типа, для чего используется следующее объявление.

```
struct date purchaseDate;
```

Или обе переменные можно объявить в одной строке, как показано ниже.

```
struct date today, purchaseDate;
```

В отличие от переменных типа `int`, `float` или `char`, при работе со структурами необходимо использовать специальный синтаксис. К членам структуры можно обратиться с помощью имени переменной, за которым следует точка и имя члена структуры. Например, для того чтобы присвоить значение 25 члену структуры `day` для структурной переменной `today`, необходимо написать следующее.

```
today.day = 25;
```

Обратите внимание, что пробелы между точкой и именами не допускаются. Для установки года необходимо написать следующее утверждение.

```
today.year = 2004;
```

Наконец, для того чтобы убедиться, что значение переменной `month` равно 12, можно написать следующее.

```
if ( today.month == 12 )
    nextMonth = 1;
```

Постарайтесь определить, что получится в результате выполнения следующих утверждений.

```
if ( today.month == 1 && today.day == 1 )
    printf ("Happy New Year!!!\n");
```

В программе из листинга 9.1 использованы новые понятия, которые обсуждались выше.

Листинг 9.1. Использование структур

```
// Программа для демонстрации использования структур
#include <stdio.h>
int main (void)
{
    struct date
    {
        int month;
        int day;
        int year;
    };
};
```

```

struct date today;

today.month = 9;
today.day = 25;
today.year = 2004;
printf ("Today's date is %i/%i/%.2i.\n", today.month, today.day,
today.year % 100);

return 0;
}

```

Листинг 9.1. Вывод

Today's date is 9/25/04.

Первое утверждение внутри процедуры `main` описывает структуру с именем `date`, которая состоит из трех целочисленных членов, названных `month`, `day` и `year`. Во втором утверждении объявляется переменная `today`, которую можно назвать структурной переменной, т.к. ей присваивается структурный тип с помощью следующего выражения: `struct date`. Для таких переменных при объявлении резервируется место в памяти для хранения трех целочисленных значений. Очень важно хорошо понимать разницу между описанием структуры и объявлением переменной структурного типа.

После объявления переменной `today` производится присваивание значений каждому из трех членов переменной `today`, как показано на рис. 9.1.

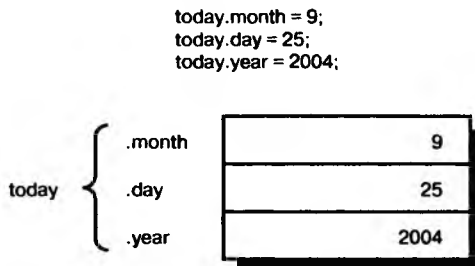


Рис. 9.1. Присваивание значений переменной структурного типа

После того как будут проведены операции присваивания, значения, содержащиеся внутри структуры, будут отображены на экране с помощью вызова утверждения `printf`. Остаток от деления значения члена структуры `today.year` на 100 будет равен 4, и именно это значение будет отображено как соответствующее году. Вспомните, что символ форматирования `%.2i` используется для вывода двух цифр и производит заполнение нулем, если цифр меньше, поэтому будет выведено "04".

Использование структур в выражениях

При вычислении выражений, к членам структур применимы те же правила языка программирования C, что и к обычным переменным. Поэтому деление целочисленного члена структуры на целое число выполняется по правилам целочисленного деления, как в следующем утверждении.

```
century = today.year / 100 + 1;
```

Предположим, вы хотите написать простую программу, в которой принимается текущая дата, а на экране отображается значение, соответствующее завтрашнему дню. На первый взгляд это кажется очень простой задачей. Вы должны попросить пользователя ввести текущую дату и затем определить значение для завтрашнего дня с помощью следующих утверждений.

```
tomorrow.month = today.month;
tomorrow.day = today.day + 1;
tomorrow.year = today.year;
```

В большинстве случаев, эти три утверждения будут хорошо работать, но в следующих двух случаях они не справятся с поставленной задачей:

- 1) если текущий день приходится на конец месяца;
- 2) если текущий день приходится на конец года (например 31 декабря).

Одним из путей решения этой проблемы может быть создание массива целых чисел, каждое из которых соответствует числу дней в каждом месяце. Значение для заданного месяца соответствует кол-ву дней в этом месяце. Утверждение

```
int daysPerMonth[12] = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30,
31 };
```

задает массив целых чисел с именем `daysPerMonth`, который содержит 12 элементов. Для каждого месяца `i` значение, содержащееся в элементе `daysPerMonth[i-1]`, соответствует кол-ву дней в этом месяце. Поэтому число дней в апреле, который является четвертым месяцем года, задается элементом массива `daysPerMonth[3]`, значение которого равно 30. (Разумеется, можно создать массив, содержащий 13 элементов, тогда не трудно сделать, чтобы номер месяца `i` указывал на соответствующий элемент `daysPerMonth[i]`. Поэтому доступ к такому массиву можно осуществлять непосредственно по номеру месяца, а не по индексу с номером месяца минус единица. Выбор того или иного способа зависит от предпочтений разработчика.)

Если заданный день выпал на конец месяца, то все, что необходимо сделать, — это добавить единицу к номеру месяца и установить для переменной `day` значение 1.

Для решения второй проблемы необходимо определить, что день является последним днем месяца и что номер месяца равен 12. В этом случае следующий день и месяц должны быть установлены в 1, а номер года увеличен на единицу.

В программе из листинга 9.2 производится запрос пользователя на введение даты и выполняется расчет даты следующего дня. Результат расчета отображается на экране.

Листинг 9.2. Определение даты следующего дня

```
// Программа определяет дату следующего дня
#include <stdio.h>
int main (void)
{
    struct date
    {
        int month;
        int day;
        int year;
    };
    struct date today, tomorrow;
```

```

const int daysPerMonth[12] = { 31, 28, 31, 30, 31, 30,
                               31, 31, 30, 31, 30, 31 };
printf ("Enter today's date (mm dd yyyy): ");
scanf ("%i%i%i", &today.month, &today.day, &today.year);
if ( today.day != daysPerMonth[today.month - 1] ) {
    tomorrow.day = today.day + 1;
    tomorrow.month = today.month;
    tomorrow.year = today.year;
}
else if ( today.month == 12 ) { // Конец года.
    tomorrow.day = 1;
    tomorrow.month = 1;
    tomorrow.year = today.year + 1;
}
else { // Конец месяца.
    tomorrow.day = 1;
    tomorrow.month = today.month + 1;
    tomorrow.year = today.year;
}
printf ("Tomorrow's date is %i/%i/%.2i.\n", tomorrow.month,
        tomorrow.day, tomorrow.year % 100);

return 0;
}

```

Листинг 9.2. Вывод

```

Enter today's date (mm dd yyyy): 12 17 2004
Tomorrow's date is 12/18/04.

```

Листинг. 9.2 Вывод (Повторение)

```

Enter today's date (mm dd yyyy): 12 31 2005
Tomorrow's date is 1/1/06.

```

Листинг. 9.2 Вывод (Второе повторение)

```

Enter today's date (mm dd yyyy): 2 28 2004
Tomorrow's date is 3/1/04.

```

Если вы внимательно посмотрите на вывод, то наверняка заметите серьезную ошибку. День после 28 февраля 2004 года указан как 1 марта 2004 года, а не как 29 февраля. Но это и не удивительно, ведь программа ничего не знает о високосных годах. Эта проблема будет решена в следующем разделе. А сейчас давайте проанализируем логику работы программы.

После описания структуры `date`, объявляются переменные `today` и `tomorrow` типа `struct date`. Затем в программе производится запрос для пользователя на введение даты. Три числа, которые будут введены, сохраняются в переменных `today.month`, `today.day` и `today.year` соответственно. Затем производится проверка того, не является ли день последним днем месяца, для чего сравниваются значения переменных `today.day` и `daysPerMonth[today.month-1]`. Если день не является последним днем

месяца, то следующий день рассчитывается путем простого добавления единицы к введенному дню, а месяц и год остаются без изменения.

Если же введенный день является последним днем месяца, то производится следующая проверка и выясняется, не является ли этот день последним днем месяца. Для этого определяется номер месяца и если он равен 12, то введенная дата соответствует 31 декабря а, следующим днем будет 1 января следующего года. Если же номер месяца не равен 12, то следующий день будет первым числом следующего месяца, а год остается тем же самым.

После того как произведен расчет даты следующего дня, это значение отображается на экране с помощью соответствующего утверждения `printf` и выполнение программы заканчивается.

Функции и структуры

Вернемся к проблеме високосного года, которая возникла в предыдущей программе. Эта программа “считает”, что в феврале всегда 28 дней, и поэтому вполне закономерно, что после введения даты 28 февраля, отображается 1 марта. Для устранения этого недостатка необходимо сделать проверку на високосный год. Если год является високосным, то в феврале должно быть 29 дней. Для получения такого результата необходимо несколько изменить работу с массивом `daysPerMonth`.

Наилучший способ внести изменения в программу листинга 9.2 — это разработать функцию с именем `numberOfDays`, которая будет рассчитывать точное кол-во дней в любом случае. Функция будет учитывать високосные года при обращении к массиву `daysPerMonth`. Все, что необходимо будет изменить в подпрограмме `main`, — это исправить утверждение `if`, в котором сравниваются значения переменных `today.day` и `daysPerMonth[today.month-1]`. Вместо этого должны сравниваться значение переменной `today.day` и значение, возвращаемое функцией `numberOfDays`.

Внимательно ознакомьтесь с программой из листинга 9.3 и постарайтесь понять, что передается в функцию `numberOfDays` в качестве аргумента.

Листинг 9.3. Улучшенная программа определения даты следующего дня

```
// Программа определения завтрашней даты.
#include <stdio.h>
#include <stdbool.h>
struct date
{
    int month;
    int day;
    int year;
};

int main (void)
{
    struct date today, tomorrow;
    int numberOfDays (struct date d);

    printf ("Enter today's date (mm dd yyyy): ");
    scanf ("%i%i%i", &today.month, &today.day, &today.year);
    if ( today.day != numberOfDays (today) ) {
        tomorrow.day = today.day + 1;
        tomorrow.month = today.month;
```

```

    tomorrow.year = today.year;
}
else if ( today.month == 12 ) { // Конец года.
    tomorrow.day = 1;
    tomorrow.month = 1;
    tomorrow.year = today.year + 1;
}
else { // Конец месяца.
    tomorrow.day = 1;
    tomorrow.month = today.month + 1;
    tomorrow.year = today.year;
}
printf ("Tomorrow's date is %i/%i/%.2i.\n", tomorrow.month,
tomorrow.day, tomorrow.year % 100);

return 0;
}
// Функция определения количества дней в месяце.
int numberOfDays (struct date d)
{
    int days;
    bool isLeapYear (struct date d);
    const int daysPerMonth[12] =
        { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
    if ( isLeapYear (d) == true && d.month == 2 )
        days = 29;
    else
        days = daysPerMonth[d.month - 1];

    return days;
}

// Функция определения високосного года.
bool isLeapYear (struct date d)
{
    bool leapYearFlag;
    if ( (d.year % 4 == 0 && d.year % 100 != 0) ||
        d.year % 400 == 0 )
        leapYearFlag = true; // Високосный год.
    else
        leapYearFlag = false; // Обычный год.

    return leapYearFlag;
}

```

Листинг 9.3. Вывод

```

Enter today's date (mm dd yyyy): 2 28 2004
Tomorrow's date is 2/29/04.

```

Листинг 9.3. Вывод (Повторение)

```

Enter today's date (mm dd yyyy): 2 28 2005
Tomorrow's date is 3/1/05.

```

Первое, на что следует обратить внимание, — это описание структуры `date` за пределами всех функций. Такое описание способствует тому, что к этой структуре могут обращаться все функции. Это аналогично поведению переменных, и если структура описана в какой-либо одной функции, то с ней можно работать только в этой функции. За пределами этой функции структура не видна. Такие структуры называются локальными. Если же описать структуру за пределами всех функций, то такая структура будет называться глобальной структурой, и ее можно использовать при объявлении типа переменных как за пределами функций, так и в самих функциях.

Внутри подпрограммы `main` прототип объявления функции

```
int numberOfDays (struct date d);
```

информирует компилятор языка C о том, что функция `numberOfDays` возвращает целое число и принимает один аргумент типа `struct date`.

Вместо сравнения значения переменной `today.day` со значением элемента множества `daysPerMonth[today.month-1]`, как это сделано в предыдущем примере, используется следующее выражение.

```
if ( today.day != numberOfDays (today) )
```

Из выражения видно, что при вызове функции в качестве аргумента в нее передается структура `today`. Внутри функции `numberOfDays` должно быть сделано соответствующее объявление для того, чтобы сообщить компилятору, что в качестве аргумента ожидается следующая структура.

```
int numberOfDays (struct date d)
```

Как и при использовании обычных переменных, но не массивов, все изменения значений для членов структуры, которые будут сделаны внутри функции, никак не отражаются на оригинальной структуре. При вызове функции в нее передается только копия оригинальной структуры, которая создается в момент вызова.

Функция `numberOfDays` начинается с определения того, является ли год високосным и является ли текущий месяц февралем. Из анализа условного выражения `if`

```
if ( isLeapYear (d) == true && d.month == 2 )
```

можно предположить, что функция `isLeapYear` возвращает `true`, если год является високосным, и `false` — в противном случае. Об этом уже упоминалось при обсуждении булевых переменных в главе 6, “Принятие решений”. Напомню, что в стандартном заголовочном файле `<stdbool.h>` объявлены переменные `bool`, `true` и `false` для удобства в работе. Именно поэтому этот файл и включен в начало листинга 9.3.

Также обратите внимание на выбор имени для функции `isLeapYear` в предыдущем выражении `if`. Это способствует лучшему пониманию условного выражения `if`, т.е. становится вполне ясно, что функция возвращает значение типа да/нет.

Продолжим рассмотрение программы. Если будет рассчитано, что месяц февраль принадлежит високосному году, то в качестве значения для переменной `days` будет установлено число 29, в противном случае значение будет выбрано из массива `daysPerMonth`. После этого значение переменной `days` возвращается в подпрограмму `main`, где продолжают вычисления, аналогичные вычислениям программы из листинга 9.2.

Функция `isLeapYear` достаточно прозрачна — здесь просто проверяется год, содержащийся в структуре `date`, которая передается в качестве аргумента, и возвращается значение `true`, если год високосный, и `false` — в противном случае.

В качестве упражнения для разработки более структурированных программ, представьте весь процесс получения завтрашней даты в виде отдельной функции. Можно

назвать эту функцию `dateUpdate` и передавать ей в качестве аргумента текущую дату. В функции будет произведен расчет завтрашней даты, которая и будет обратно возвращена в вызвавшую ее подпрограмму. В листинге 9.4 показано, как это можно сделать на языке программирования C.

Листинг 9.4. Вариант программы определения завтрашней даты, версия 2

```
// Программа определения завтрашней даты.
#include <stdio.h>
#include <stdbool.h>

struct date
{
    int month;
    int day;
    int year;
};

// Функция расчета завтрашней даты.
struct date dateUpdate (struct date today)
{
    struct date tomorrow;
    int numberOfDays (struct date d);

    if ( today.day != numberOfDays (today) ) {
        tomorrow.day = today.day + 1;
        tomorrow.month = today.month;
        tomorrow.year = today.year;
    }
    else if ( today.month == 12 ) {      // Конец года.
        tomorrow.day = 1;
        tomorrow.month = 1;
        tomorrow.year = today.year + 1;
    }
    else {                              // Конец месяца.
        tomorrow.day = 1;
        tomorrow.month = today.month + 1;
        tomorrow.year = today.year;
    }
    return tomorrow;
}

// Функция расчета кол-ва дней в месяце.
int numberOfDays (struct date d)
{
    int days;
    bool isLeapYear (struct date d);
    const int daysPerMonth[12] =
        { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };

    if ( isLeapYear && d.month == 2 )
        days = 29;
    else
        days = daysPerMonth[d.month - 1];
    return days;
}
```



```
// Функция определения високосного года.
bool isLeapYear (struct date d)
{
    bool leapYearFlag;

    if ( (d.year % 4 == 0 && d.year % 100 != 0) ||
        d.year % 400 == 0 )
        leapYearFlag = true; // It's a leap year
    else
        leapYearFlag = false; // Not a leap year
    return leapYearFlag;
}

int main (void)
{
    struct date dateUpdate (struct date today);
    struct date thisDay, nextDay;

    printf ("Enter today's date (mm dd yyyy): ");
    scanf ("%i%i%i", &thisDay.month, &thisDay.day,
        &thisDay.year);
    nextDay = dateUpdate (thisDay);
    printf ("Tomorrow's date is %i/%i/%.2i.\n", nextDay.month,
        nextDay.day, nextDay.year % 100);

    return 0;
}
```

Листинг 9.4. Вывод

```
Enter today's date (mm dd yyyy): 2 28 2008
Tomorrow's date is 2/29/08.
```

Листинг 9.4. Вывод (Повторение)

```
Enter today's date (mm dd yyyy): 2 22 2005
Tomorrow's date is 2/23/05.
```

Внутри подпрограммы main, утверждение

```
next_date = dateUpdate (thisDay);
```

демонстрирует возможность передавать структуру в функцию и возвращать ее из функции. Функция dateUpdate объявлена соответствующим образом, и из объявления видно, что она возвращает значение типа struct date. Код внутри функции тот же самый, что и в подпрограмме main из листинга 9.3. Функции numberOfDays и isLeapYear остаются неизменными.

Еще раз убедитесь, что вы хорошо понимаете иерархию вызовов функций в предыдущей программе. Функция main вызывает функцию dateUpdate, которая в свою очередь вызывает функцию numberOfDays, которая тоже производит вызов функции с именем isLeapYear.

Структура для хранения значений времени

Предположим, вам необходимо хранить в программе различные значения времени в часах, минутах и секундах. Поскольку вы уже знаете, как использовать структуру `date`, в которой логически сгруппированы день, месяц и год, то не должно возникать никаких затруднений при объявлении структуры для хранения значений времени, в которой сгруппированы часы, минуты и секунды. Пример объявления такой структуры приведен ниже.

```
struct time
{
    int hour;
    int minutes;
    int seconds;
};
```

Большинство пользователей выбирают формат отображения времени в виде 24 различных часовых значений; он известен как военный формат времени. Такое представление позволяет избежать дополнительных уточняющих префиксов наподобие "a.m." или "p.m.". Отсчет времени начинается с 0 часов в полночь и увеличивается на единицу, пока не достигнет 23 часов, что означает 11:00 p.m. При этом, например, 4:30 означает 4:30 a.m., а 16:30 означает 4:30 p.m. Время 12:00 представляет полдень, тогда как 00:01 означает минуту после полночи.

Обычно все компьютеры имеют часы, которые всегда включены и производят отсчет времени. Такие часы необходимы для информирования пользователя о текущем времени, для автоматического вызова программ в определенное время, для записи значений времени происходящих событий и т.д. Обычно несколько программ всегда связаны с часами. Некоторые программы могут выполняться каждую секунду, например, обновление текущего времени, которое хранится в отдельном участке памяти и используется в различных ситуациях.

Предположим, вы хотите имитировать программу, описанную выше, в частности, разработать программу, которая обновляет время каждую секунду. Если вы **подумаете** на этом хотя бы секунду (это я хотел скаламбурить), то поймете, что эта задача может быть решена аналогично тому, как эта проблема была решена для получения даты завтрашнего дня.

Так же, как и при расчете даты следующего дня, необходимо учитывать специальные требования. Здесь необходимо производить обновление времени на основе некоторого алгоритма. То есть должны быть учтены следующие требования:

1. Если кол-во секунд достигает 60, то счетчик секунд должен быть установлен в 0, а значение минут должно увеличиться на 1.
2. Если кол-во минут достигает 60, то счетчик минут должен быть установлен в 0, а значение для количества часов должно увеличиться на 1.
3. Если кол-во часов достигает 24, то счетчики секунд, минут и часов должны быть установлены в 0.

В программе из листинга 9.5 используется функция с именем `timeUpdate`, которая принимает в качестве аргумента текущее время и возвращает значение времени, которое должно быть через одну секунду.

Листинг 9.5. Ежесекундное обновление времени

```
// Программа для ежесекундного обновления времени.
#include <stdio.h>
struct time
{
    int hour;
    int minutes;
    int seconds;
};

int main (void)
{
    struct time timeUpdate (struct time now);
    struct time currentTime, nextTime;
    printf ("Enter the time (hh:mm:ss): ");
    scanf ("%i:%i:%i", &currentTime.hour,
            &currentTime.minutes, &currentTime.seconds);
    nextTime = timeUpdate (currentTime);
    printf ("Updated time is %.2i:%.2i:%.2i\n", nextTime.hour,
            nextTime.minutes, nextTime.seconds );
    return 0;
}

// Функция для ежесекундного обновления времени.
struct time timeUpdate (struct time now)
{
    ++now.seconds;
    if ( now.seconds == 60 ) {        // Следующая минута.
        now.seconds = 0;
        ++now.minutes;
        if ( now.minutes == 60 ) {    // Следующий час.
            now.minutes = 0;
            ++now.hour;
            if ( now.hour == 24 )      // Полночь.
                now.hour = 0;
        }
    }
    return now;
}
```

Листинг 9.5. Вывод

```
Enter the time (hh:mm:ss): 12:23:55
Updated time is 12:23:56
```

Листинг 9.5. Вывод (Повторение)

```
Enter the time (hh:mm:ss): 16:12:59
Updated time is 16:13:00
```

Листинг 9.5. Вывод (Второе повторение)

```
Enter the time (hh:mm:ss): 23:59:59
Updated time is 00:00:00
```

Подпрограмма `main` запрашивает пользователя на ввод времени. Функция `scanf` для считывания данных использует следующую строку форматирования.

```
"%i:%i:%i"
```

Использование неформатного символа, такого как “:”, в строке форматирования указывает функции `scanf` на то, что ожидается очередной символ с устройства ввода. Поэтому такая строка форматирования, как показано выше, необходима для ввода трех целочисленных значений, где первое число отделяется от второго двоеточием, и второе число отделяется от третьего тоже двоеточием. В главе 16 “Операции ввода и вывода в языке C” вы узнаете больше о том, как функция `scanf` производит обработку и контроль вводимых значений и какие значения она возвращает.

После того, как введено время, в программе вызывается функция `timeUpdate`, в которую в качестве аргумента передается значение переменной `currentTime`. Результат расчета присваивается переменной `nextTime` типа `struct time`, значение которой отображается с помощью функции `printf`.

Функция `timeUpdate` начинает вычисления с увеличения времени на одну секунду, после чего производится сравнение общего кол-ва секунд со значением 60. Если это равенство выполняется, то счетчик кол-ва секунд устанавливается в 0, а значение кол-ва минут увеличивается на 1. Следующая проверка производится для кол-ва минут, которое не должно превышать или должно быть равно 60. Если кол-во минут равно 60, то счетчик минут устанавливается в 0 и увеличивается число часов. Наконец, если в обоих предыдущих случаях выполнялось равенство, то производится проверка кол-ва часов, которое не должно превышать или быть равно 24. Если число часов равно 24, то производится обнуление счетчика часов. Затем функция возвращает в вызвавшую подпрограмму значение переменной `now`, которое соответствует времени, наступившему через одну секунду.

Инициализация структур

Инициализация структур аналогична инициализации массивов — значения элементов просто перечисляются внутри фигурных скобок, и при этом каждое значение отделяется запятой.

Для инициализации структурной переменной `today` типа `date` значениями July 2, 2005, можно использовать следующее утверждение.

```
struct date today = { 7, 2, 2005 };
```

Утверждение

```
struct time this_time = { 3, 29, 55 };
```

устанавливает переменную `this_time` типа `struct time` и задает для нее время 3:29:55 a.m. Как и для других переменных, если структура `this_time` является локальной структурной переменной, она инициализируется каждый раз при вызове функции. Если структурную переменную сделать статической, т.е. поместить ключевое слово `static` перед объявлением структуры, она будет инициализироваться только один раз при первом выполнении функции. В любом случае, инициализирующие значения, перечисляемые внутри фигурных скобок, должны быть константными выражениями. Как и при инициализации массива, можно указывать не все значения входящих элементов. Поэтому утверждение

```
struct time time1 = { 12, 10 };
```

задает значение для элемента `time1.hour` равным 12, и для `time1.minutes` равным 10, но не задает значение для элемента структуры `time1.seconds`. В этом случае значение будет неопределенным.

Можно задавать значения для элементов структуры, перечисляя их имена с соответствующими значениями. Общий формат в таком случае будет следующим.

```
.member = value
```

При этом вы можете инициализировать элементы в любом порядке или инициализировать отдельные элементы. Например, в утверждении

```
struct time time1 = { .hour = 12, .minutes = 10 };
```

переменная `time1` будет инициализирована теми же значениями, что и в предыдущем случае. Утверждение

```
struct date today = { .year = 2004 };
```

задаст для структурной переменной `today` значение года равным 2004.

Составные литералы

Можно присвоить одно или несколько значений элементам структуры в одном утверждении, используя т.н. *составные литералы*. Например, полагая, что переменная `today` уже объявлена как переменная типа `struct date`, можно выполнить присваивание элементам таким образом, как показано в программе из листинга 9.1, или выполнить присваивание в одном утверждении, как показано ниже.

```
today = (struct date) { 9, 25, 2004 };
```

Причем такое утверждение может появиться в любом месте программы – это не является объявлением. Оператор приведения типа используется для того, чтобы сообщить компилятору тип выражения, который в данном случае должен соответствовать `struct date`, а само выражение представляет список значений в строго заданном порядке. Значения должны быть перечислены в том же порядке, как и при инициализации структурной переменной.

Можно использовать и имена элементов для присваивания значений, как это показано ниже.

```
today = (struct date) { .month = 9, .day = 25, .year = 2004 };
```

Преимущество такого способа заключается в том, что значения могут перечисляться в любом порядке. Без явного указания имен элементов они должны находиться в том порядке, в котором находятся элементы структуры. В примере ниже показана функция `dateUpdate` из программы 9.4, переписанная с использованием преимуществ составных литералов.

```
// Функция для расчета завтрашней даты – использование составных
литералов.
struct date dateUpdate (struct date today)
{
    struct date tomorrow;
    int numberOfDays (struct date d);
    if (*today.day != numberOfDays (today) )
        tomorrow = (struct date) { today.month, today.day + 1, today.
year };
```

```

    else if ( today.month == 12 ) // Конец
года.
    tomorrow = (struct date) { 1, 1, today.year + 1 };
    else // Конец
месяца.
    tomorrow = (struct date) { today.month + 1, 1, today.year };

    return tomorrow;
}

```

То, будете ли вы использовать составные литералы, зависит только от вас. В данном случае использование составных литералов делает функцию `dateUpdate` более компактной и удобной для чтения. Составные литералы можно использовать и в других местах, где необходимо задавать значения элементам структур. Ниже приводится допустимый хотя и не всегда практичный пример такого использования.

```

nextDay = dateUpdate ((struct date) { 5, 11, 2004 } );

```

В функцию `dateUpdate` в качестве аргумента передается составной литерал, который представляет тип `struct date`. Такой тип является параметром данной функции.

Массивы структур

Вы уже могли убедиться, как полезны структуры для логического группирования связанных элементов. Например, используя структуру `time`, в программе можно работать только с одной переменной вместо трех, которые используются для хранения значений времени. Таким образом, используя в программе 10 различных временных отсчетов, вы будете иметь дело только с 10-ю переменными вместо 30.

Но еще лучшим способом для хранения 10 различных временных отсчетов будет комбинация таких мощных средств языка программирования C, как структуры и массивы. Язык C разрешает хранить в массивах не только простые типы данных, но и задавать массивы структур. Например, в утверждении

```

struct time experiments[10];

```

объявляется массив с именем `experiments`, который состоит из 10 элементов. Каждый элемент внутри массива имеет тип `struct time`. Аналогично, объявление

```

struct date birthdays[15];

```

задает массив с именем `birthdays`, который содержит 15 элементов типа `struct date`. Ссылаться на элементы массива можно с помощью уже привычного правила. Для того чтобы установить во втором элементе массива день рождения August 8, 1986, можно использовать следующую последовательность утверждений.

```

birthdays[1].month = 8;
birthdays[1].day = 8;
birthdays[1].year = 1986;

```

Для того чтобы передать в функцию `checkTime` структуру `time`, содержащуюся в массиве, можно применить следующее утверждение.

```

checkTime (experiments[4]);

```

При этом необходимо понимать, что при объявлении функции `checkTime` в качестве параметра был указан тип `struct time`.

```
void checkTime (struct time t0)
{
    .
    .
    .
}
```

Инициализация массивов, содержащих структуры, подобна инициализации многомерных массивов. Поэтому в следующем утверждении

```
struct time runTime [5] =
{ {12, 0, 0}, {12, 30, 0}, {13, 15, 0} };
```

будут заданы три первых временных отсчета в массиве `runTime` — это время 12:00:00, 12:30:00 и 13:15:00. Внутренние пары фигурных скобок не являются обязательными и предыдущее утверждение можно написать следующим образом.

```
struct time runTime[5] =
{ 12, 0, 0, 12, 30, 0, 13, 15, 0 };
```

Следующее утверждение

```
struct time runTime[5] =
{ [2] = {12, 0, 0} };
```

инициализирует только третий элемент массива заданными значениями, тогда как в утверждении

```
static struct time runTime[5] = { [1].hour = 12, [1].minutes = 30 };
```

устанавливаются только часы и минуты второго элемента массива `runTime`, которые принимают значения 12 и 30 соответственно.

В программе из листинга 9.6 задается массив структур времени с именем `testTimes`. Затем в программе вызывается функция `timeUpdate`, которая уже описана в программе 9.5. Из-за ограничений по объему функция `timeUpdate` не включена в листинг программы. Однако в приведенном комментарии указано место, где должна быть вставлена эта функция.

В программе из листинга 9.6 задается массив с именем `testTimes` для хранения пяти временных отсчетов. Элементам этого массива присваиваются значения, которые содержат временные отсчеты 11:59:59, 12:00:00, 1:29:59, 23:59:59 и 19:12:27 соответственно. На рис 9.2 можно увидеть, как элементы массива размещаются в памяти компьютера. К отдельной структуре `time` можно получить доступ с помощью соответствующего индекса от 0 до 4. Доступ к членам структуры (`hour`, `minutes` и `seconds`) производится с указанием имени члена, отделенного от имени элемента массива точкой.

В программе из листинга 9.6 для каждого элемента массива `testTimes` вызывается функция обновления времени `timeUpdate` и затем производится отображение обновленного времени на экране.

Листинг 9.6. Демонстрация использования массива структур

```
// Программа демонстрирует использование массива структур
#include <stdio.h>
struct time
{
    int hour;
    int minutes;
    int seconds;
};

int main (void)
{
    struct time timeUpdate (struct time now);
    struct time testTimes[5] =
        { { 11, 59, 59 }, { 12, 0, 0 }, { 1, 29, 59 },
          { 23, 59, 59 }, { 19, 12, 27 } };
    int i;
    for ( i = 0; i < 5; ++i ) {
        printf ("Time is %i:%i:%i", testTimes[i].hour,
            testTimes[i].minutes, testTimes[i].seconds);
        testTimes[i] = timeUpdate (testTimes[i]);
        printf (" ...one second later it's %i:%i:%i\n",
            testTimes[i].hour, testTimes[i].minutes, testTimes[i].seconds);
    }
    return 0;
}
// ***** Функция timeUpdate должна находиться здесь *****
```

Листинг 9.6. Вывод

```
Time is 11:59:59 ...one second later it's 12:00:00
Time is 12:00:00 ...one second later it's 12:00:01
Time is 01:29:59 ...one second later it's 01:30:00
Time is 23:59:59 ...one second later it's 00:00:00
Time is 19:12:27 ...one second later it's 19:12:28
```

Концепция массивов структур является очень мощной и важной в языке программирования C. Поэтому проверьте себя и убедитесь, что вы хорошо понимаете все положения этой концепции, прежде чем двигаться дальше.

Структуры, содержащие структуры

Язык программирования C обеспечивает огромное количество вариантов использования структур. Например, можно задать структуру, которая в свою очередь включает другие структуры в качестве одного или нескольких своих членов, или объявить структуру, которая содержит массивы.

Вы уже имели возможность видеть, как можно логически сгруппировать месяц, день и год в одной структуре с именем `date` и как объединить часы, минуты и секунды в структуре с именем `time`. Возможно, в некоторых случаях вам понадобится логически объединить дату и время. Например, вам нужно составить список событий, которые происходят в некоторые моменты времени.

testTimes[0]	{	.hour	11
		.minutes	59
		.seconds	59
testTimes[1]	{	.hour	12
		.minutes	0
		.seconds	0
testTimes[2]	{	.hour	1
		.minutes	29
		.seconds	59
testTimes[3]	{	.hour	23
		.minutes	59
		.seconds	59
testTimes[4]	{	.hour	19
		.minutes	12
		.seconds	27

Рис. 9.2. Расположение массива testTimes в памяти

Исходя из предыдущих дискуссий, вы уже знаете, что в этом случае вам необходим удобный доступ и одновременный и к дате, и ко времени. Для этого вы можете описать новую структуру, например, с именем dateAndTime, которая в качестве своих членов включает два элемента, дату и время.

```
struct dateAndTime
{
    struct date sdate;
    struct time stime;
};
```

Первый член этой структуры представляет тип struct date и называется sdate. Второй член структуры dateAndTime имеет имя stime и тип struct time.

При этом необходимо, чтобы описания структур date и time были сделаны до того, как будет описана структура dateAndTime. Объявление переменной типа struct dateAndTime можно сделать следующим образом

```
struct dateAndTime event;
```

Для ссылки на структуру date из переменной event, используется привычный синтаксис.

```
event.sdate
```

Поэтому вы можете вызвать функцию dateUpdate с датой в качестве аргумента и присвоить результат той же структуре, которую использовали в качестве аргумента, как показано ниже.

```
event.sdate = dateUpdate (event.sdate);
```

То же самое вы можете сделать со структурой `time`, содержащейся в структуре `dateAndTime`.

```
event.stime = timeUpdate (event.stime);
```

Для ссылки на отдельный член одной из внутренних структур, используется точка, которая ставится после имени структуры и перед именем члена.

```
event.sdate.month = 10;
```

В этом утверждении устанавливается месяц `October` для члена `month` структуры `date`, содержащейся в структуре `event`. В следующем выражении

```
++event.stime.seconds;
```

добавляется секунда к члену `seconds`, содержащемуся в структуре `time`.

Переменная `event` может быть инициализирована уже привычным способом.

```
struct dateAndTime event =  
{ { 2, 1, 2004 }, { 3, 30, 0 } };
```

В этом случае для переменной `event` устанавливается дата `February 1, 2004` и время `3:30:00`.

Разумеется, при инициализации вы можете использовать имена членов, как показано ниже.

```
struct dateAndTime event =  
{ { .month = 2, .day = 1, .year = 2004 },  
  { .hour = 3, .minutes = 30, .seconds = 0 }  
};
```

Также просто создать массив структур `dateAndTime`, что и показано ниже.

```
struct dateAndTime events[100];
```

В данном утверждении объявляется массив с именем `events`, который содержит 100 элементов типа `struct dateAndTime`. На четвертую структуру массива `events` типа `dateAndTime` можно сослаться как обычно `events[3]`, а i -тая дата массива может быть передана в функцию `dateUpdate` следующим образом.

```
events[i].sdate = dateUpdate (events[i].sdate);
```

Для того чтобы для первого элемента массива установить время в полночь, можно использовать следующую последовательность утверждений.

```
events[0].stime.hour = 12;  
events[0].stime.minutes = 0;  
events[0].stime.seconds = 0;
```

Структуры, содержащие массивы

Как понятно из заголовка этого раздела, можно задавать структуры, которые содержат массивы в качестве своих членов. В большинстве случаев это делается для задания массива символов внутри структуры. Например, предположим, что вам необходимо задать структуру с именем `month`, которая в качестве членов использует порядковый

номер месяца и аббревиатуру для названия месяца. Это сделано в приведенном ниже описании структуры.

```
struct month
{
    int numberOfDays;
    char name[3];
};
```

При этом задается структура, которая содержит целочисленный член с именем `numberOfDays` и член типа `char` с именем `name`. Член `name` представляет массив из трех символов. После описания структуры вы можете объявить переменную типа `struct month` обычным образом.

```
struct month aMonth;
```

Вы можете присвоить значение `January` членам структуры `aMonth` с помощью следующей последовательности утверждений.

```
aMonth.numberOfDays = 31;
aMonth.name[0] = 'J';
aMonth.name[1] = 'a';
aMonth.name[2] = 'n';
```

Или вы можете инициализировать эту переменную теми же самыми значениями следующим образом:

```
struct month aMonth = { 31, { 'J', 'a', 'n' } };
```

Продолжая следовать этой логике вы можете задать 12 структур типа `month` в качестве элементов массива, каждая из которых будет определять отдельный месяц.

```
struct month months[12];
```

В программе из листинга 9.7 используется массив `months`, который сначала инициализируется необходимыми значениями, и затем эти значения отображаются на экране.

Возможно, вам будет легче понять ссылки на отдельные элементы массива `months`, который используется в программе, если вы предварительно ознакомитесь с рис. 9.3.

Листинг 9.7. Демонстрация использования структур и массивов

```
// Программа демонстрирует использование структур и массивов
#include <stdio.h>
int main (void)
{
    int i;
    struct month
    {
        int numberOfDays;
        char name[3];
    };
    const struct month months[12] =
        { { 31, {'J', 'a', 'n'} }, { 28, {'F', 'e', 'b'} },
          { 31, {'M', 'a', 'r'} }, { 30, {'A', 'p', 'r'} },
          { 31, {'M', 'a', 'y'} }, { 30, {'J', 'u', 'n'} },
          { 31, {'J', 'u', 'l'} }, { 31, {'A', 'u', 'g'} },
          { 30, {'S', 'e', 'p'} }, { 31, {'O', 'c', 't'} },
```

```

    { 30, {'N', 'o', 'v'} }, { 31, {'D', 'e', 'c'} } } };

printf ("Month Number of Days\n");
printf ("-----\n");
for ( i = 0; i < 12; ++i )
    printf ( " %c%c%c %i\n",
        months[i].name[0], months[i].name[1],
        months[i].name[2], months[i].numberOfDays );

return 0;
}

```

Листинг 9.7. Вывод

Month Number of Days

Jan	31
Feb	28
Mar	31
Apr	30
May	31
Jun	30
Jul	31
Aug	31
Sep	30
Oct	31
Nov	30
Dec	31

Как можно видеть на рис. 9.3, выражение вида

```
months[0]
```

ссылается на всю структуру month, расположенную на первом месте в массиве months. Тип этого выражения — struct month. Следовательно, когда производится передача элемента массива months[0] в функцию в качестве аргумента, соответствующий формальный параметр в функции должен быть объявлен как тип struct month. Исходя из этого, выражение

```
months[0].numberOfDays
```

будет ссылаться на член numberOfDays структуры month, которая содержится в элементе массива months[0]. Тип этого выражения — int. Выражение

```
months[0].name
```

будет ссылаться на массив из трех символов с именем name, находящийся в структуре month, содержащейся в элементе массива months[0].

Если передавать это выражение в качестве выражения в функцию, то соответствующий формальный параметр должен быть объявлен как массив типа char. Наконец, выражение

```
months[0].name[0]
```

ссылается на первый символ массива с именем name, содержится в элементе months[0], это символ "J".

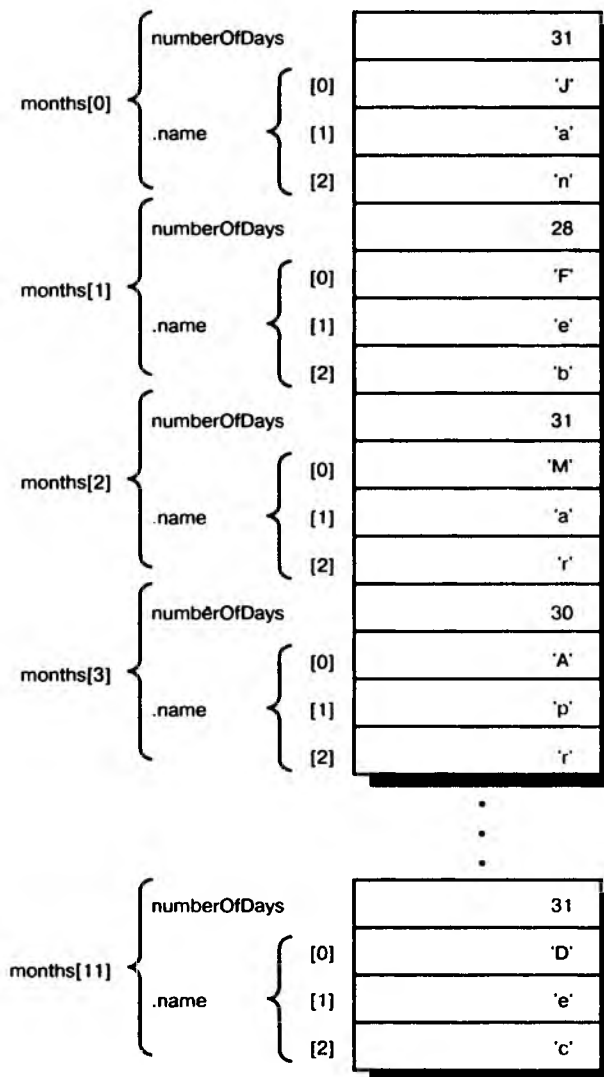


Рис. 9.3. Массив months

Варианты структур

При описании структур вы можете использовать различные варианты. Во-первых, одновременно с описанием структуры можно объявить переменную. Это можно сделать, вставив имя или имена переменных до того, как будут поставлены заключительные точка с запятой. Например, утверждение

```
struct date
{
    int month;
    int day;
```

```
int year;
} todaysDate, purchaseDate;
```

описывает структуру date и одновременно объявляет переменные todaysDate и purchaseDate, которые принимают тип date. При этом можно даже инициализировать переменные аналогично тому, как это делается в отдельном утверждении.

```
struct date
{
    int month;
    int day;
    int year;
} todaysDate = { 1, 11, 2005 };
```

Здесь задана структура date и объявлена переменная todaysDate, которая инициализируется соответствующими значениями. Если при описании структуры объявляются все необходимые переменные, то имя структуры можно пропустить, как это сделано в утверждении ниже, где объявлен массив структур dates, состоящий из 100 элементов.

```
struct
{
    int month;
    int day;
    int year;
} dates[100];
```

Каждый элемент массива представляет структуру, состоящую из трех членов: month, day и year. Поскольку имя структуры не указано, то при необходимости впоследствии объявить переменную такого же типа, необходимо снова объявить явно структуру.

В этой главе вы познакомились с тем, как можно объединить взаимосвязанные переменные и получить преимущества от использования унифицированного доступа. Вы также увидели, как легко создавать массивы структур и передавать элементы массива в функции. В следующей главе вы более подробно познакомитесь с тем, как работать с массивами символов, которые также называют символьными строками.

Упражнения

1. Наберите и выполните все семь программ, представленных в данной главе. Сравните вывод, сделанный каждой программой, с выводом, показанным в книге для каждой программы.
2. В определенных приложениях, особенно касающихся финансовой сферы, необходимо часто рассчитывать кол-во дней между двумя датами. Например, кол-во дней между July 2, 2005 и July 16, 2005 очевидно равно 14. Но чему равно число дней между August 8, 2004 и February 22, 2005? Этот расчет будет немного сложнее.

Для такого случая выведена формула, которая позволяет облегчить расчет кол-ва дней между двумя датами.

$$N = 1461 * f(\text{year}, \text{month}) / 4 + 153 * g(\text{month}) / 5 + \text{day}$$

где

$f(\text{year}, \text{month}) = \text{year} - 1$ если $\text{month} \leq 2$

иначе year

$g(\text{month}) = \text{month} + 13$ если $\text{month} \leq 2$

иначе $\text{month} + 1$

Для примера рассчитаем по этой формуле кол-во дней, прошедших между August 8, 2004 и February 22, 2005. Это можно легко сделать, подставив соответствующие значения в формулу и вычитая значение N2 из N1, как это показано ниже.

$$N1 = 1461 * f(2004, 8) / 4 + 153 * g(8) / 5 + 3$$

$$= (1461 * 2004) / 4 + (153 * 9) / 5 + 3$$

$$= 2,927,844 / 4 + 1,377 / 5 + 3$$

$$= 731961 + 275 + 3$$

$$= 732239$$

$$N2 = 1461 * f(2005, 2) / 4 + 153 * g(2) / 5 + 21$$

$$= (1461 * 2004) / 4 + (153 * 15) / 5 + 21$$

$$= 2,927,844 / 4 + 2295 / 5 + 21$$

$$= 731961 + 459 + 21$$

$$= 732441$$

$$\text{Кол-во прошедших дней} = N2 - N1$$

$$= 732441 - 732239$$

$$= 202$$

В результате расчета получим кол-во дней между датами, равное 202. Используемая формула применима ко всем датам после March 1, 1900 (единица должна быть добавлена к N для всех дат от March 1, 1800 до February 28, 1900, и двойка — для дат между March 1, 1700 и February 28, 1800).

Напишите программу, которая позволяет пользователю ввести две даты и получить количество дней между ними. Постарайтесь выделить в программе логические блоки и структурировать программу соответствующим образом. Например, вы должны иметь функцию, которая принимает в качестве аргумента структуру `date` и возвращает значение N, рассчитанное по вышеуказанной формуле. Эта функция будет вызываться дважды, и разность между возвращаемыми значениями будет составлять необходимое кол-во дней.

3. Напишите функцию `elapsed_time`, которая принимает в качестве аргументов две структуры `time` и возвращает структуру `time`, которая является разностью между введенными значениями времени в часах, минутах и секундах. Поэтому вызвав функцию

```
elapsed_time (time1, time2)
```

где `time1` имеет значение 3:45:15, а `time2` представляет 9:44:03, вы должны получить структуру `time`, которая будет содержать 5 часов, 58 минут и 48 секунд.

4. Если вы возьмете значение переменной `N`, рассчитанное в упражнении 2, вычтете из него число 621049 и разделите по модулю 7, вы получите число от 0 до 6, которое будет представлять день недели (от воскресенья до субботы соответственно), на который выпадает введенная дата. Например, рассчитанным значением `N` для August 8, 2004 будет 732239. Разность между 732,239 и 621,049 даст 111190, а $111190 \% 7$ дает 2, что свидетельствует о том, что это вторник (Tuesday).
Используйте функцию, разработанную в предыдущем примере, для разработки программы, которая будет отображать дни недели на английском языке (например "Monday").
5. Напишите функцию с именем `clockKeeper`, которая принимает в качестве аргумента структуру `dateAndTime`, описанную в этой главе. Функция должна в свою очередь вызывать функцию `timeUpdate`, и если время достигает полночи, функция должна вызвать функцию `dateUpdate` для указания следующего дня. Функция должна возвращать обновленную структуру `dateAndTime`.
6. Замените функцию из программы 9.4 модифицированной функцией, которая использует составные литералы. Запустите программу для проверки специфических операций.

10

Символьные строки

У вас уже должно быть достаточно знаний, чтобы перейти к более детальному изучению символьных строк. Первое знакомство с символьными строками произошло в главе 3, “Компиляция и запуск первой программы”, когда вы писали свою первую программу. В утверждении

```
printf ("Programming in C is fun.\n");
```

аргумент, передаваемый в функцию `printf`, является символьной строкой

```
"Programming in C is fun.\n"
```

Двойные кавычки используются для выделения символьной строки, которая может содержать любые комбинации букв, чисел или специальных символов, отличных от двойных кавычек. Но как вы скоро увидите, двойные кавычки также можно включать в символьную строку, но с небольшим дополнением.

Когда вы познакомились с типом `char`, вы узнали, что переменная, которая объявлена как тип `char`, должна содержать только один символ. При присваивании значения такой переменной, символ должен быть заключен в одинарные кавычки. Таким образом, присваивание

```
plusSign = '+';
```

приведет к тому, что переменная `plusSign`, которая должна быть предварительно объявлена как тип `char`, будет содержать символ `“+”`. Также вы узнали, что необходимо делать различие между двойными и одинарными кавычками. Поэтому следующее присваивание будет некорректным

```
plusSign = "+";
```

Хорошо запомните, что двойные и одинарные кавычки в языке программирования C используются для получения различных типов констант.

Символьные массивы

Если вы хотите работать с переменными, которые содержат более одного символа, то вам не обойтись без символьных массивов. В программе из листинга 7.6 вы уже видели символьный массив с именем `word`, который был объявлен следующим образом.

```
char word [] = { 'H', 'e', 'l', 'l', 'o', '!' };
```

Вспомните, что при отсутствии точного указания размера массива, компилятор языка C автоматически подсчитывает количество элементов массива, используя количество задаваемых для инициализации значений. При этом в нашем случае будет выделено место точно для шести символов, как показано на рис. 10.1.

Вспомните, что тип `wchar_t` применяется для хранения т.н. "широких символов", которые используются для обработки отдельных символов из интернационального набора символов. В данной главе обсуждаются способы хранения последовательностей отдельных символов.

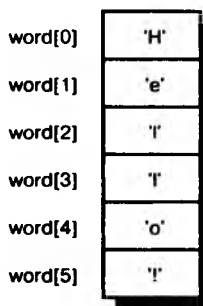


Рис. 10.1. Размещение в памяти массива с именем `word`

Для распечатки содержимого массива `word`, вы должны выделить каждый элемент массива и отобразить его с помощью символа форматирования `%c`.

Используя аналогичные приемы, вы можете создать несколько полезных функций для работы с символьными строками. Наиболее часто используемые операции с символьными строками – это объединение двух символьных строк (concatenation), копирование одной строки в другую (copy), извлечение части символьной строки (substring) и равенство двух символьных строк (последовательности символов одинаковые). Возьмем первую операцию, т.е. объединение (конкатенацию) строк, и разработаем функцию, которая будет выполнять эту операцию. Назовем эту функцию `concat`, и обращение к ней будет производиться следующим образом.

```
concat (result, str1, n1, str2, n2);
```

Здесь параметры `str1` и `str2` представляют два символьных массива, которые должны быть объединены, а параметры `n1` и `n2` представляют число элементов соответствующего массива. При этом функция получается достаточно гибкой и с ее помощью можно объединять массивы различной длины. Параметр `result` представляет результирующий символьный массив, который будет содержать как элементы массива `str1`, так и элементы массива `str2`. Элементы массива `str2` будут располагаться за элементами массива `str1`. Соответствующая программа приведена в листинге 10.1.

Листинг 10.1. Объединение символьных массивов

```
// функция для объединения символьных массивов
#include <stdio.h>
void concat (char result[], const char str1[], int n1,
             const char str2[], int n2)
{
    int i, j;
    // Копирование символов массива str1 в result.
    for ( i = 0; i < n1; ++i )
```

```

    result[i] = str1[i];
    // Копирование символов массива str2 в result.
    for ( j = 0; j < n2; ++j )
        result[n1 + j] = str2[j];
}

int main (void)
{
    void concat (char result[], const char str1[], int n1,
        const char str2[], int n2);
    const char s1[5] = { 'T', 'e', 's', 't', ' ' };
    const char s2[6] = { 'w', 'o', 'r', 'k', 's', '.' };
    char s3[11];
    int i;

    concat (s3, s1, 5, s2, 6);
    for ( i = 0; i < 11; ++i )
        printf ("%c", s3[i]);
    printf ("\n");
    return 0;
}

```

Листинг 10.1. Вывод

Test works.

В первом цикле for внутри функции concat производится копирование символов из символьного массива str1 в массив result. В этом цикле выполняется n1 итераций, где n1 задает число символов, содержащихся в массиве str1.

Во втором цикле for внутри функции concat также производится копирование символов — но уже из символьного массива str2 в массив result. Поскольку массив str1 содержит n1 символов, которые были скопированы в массив result, то элемент result[n1] будет содержать символ, непосредственно следующий за элементами массива str1. После завершения второго цикла for, массив result будет содержать n1+n2 символов, представляющих строку str2, добавленную к строке str1.

В подпрограмме main заданы две (s1 и s2) переменные типа const character. Первый символьный массив инициализируется символами 'T', 'e', 's', 't' и ' '. Последний символ представляет пробел и является допустимой символьной константой. Второй символьный массив инициализируется символами 'w', 'o', 'r', 'k', 's' и '.'. Третий символьный массив s3 должен объявляться с размером, достаточным для включения двух этих символьных массивов, т.е. 11 элементов. Поэтому он объявляется без ключевого слова const, ведь его размеры нельзя определить заранее.

В следующем утверждении.

```
concat (s3, s1, 5, s2, 6);
```

производится вызов функции concat для объединения двух символьных массивов s1 и s2 в массиве s3. Аргументы 5 и 6 передаются в функцию для указания количества символов в массивах s1 и s2 соответственно.

После того как функция concat закончит работу и управление возвратится в функцию main, вызывается цикл for для отображения результатов работы функции concat. Из массива s3 на экране будет отображено 11 символов. Исходя из того, что

отобразится на экране, можно предположить, что функция `concat` работает правильно. Но никогда не задавайте размеры массива `s3` подобным образом. Небольшая ошибка в размерах предыдущих массивов может привести к непредсказуемым результатам при выполнении программы.

Символьные строки переменной длины

Вы можете использовать подход, примененный в функции `concat`, для разработки других функций, обрабатывающих символьные массивы. То есть вы можете разработать несколько подпрограмм, каждая из которых принимает в качестве аргументов один или несколько символьных массивов плюс числа, соответствующие размерам каждого символьного массива. Но поработав некоторое время с такими функциями, вы обнаружите, что работать с ними крайне утомительно, т.к. приходится помнить размеры массивов и подставлять их значения в функцию. И будет совсем неудобно, если использовать строковые массивы переменной длины. Во избежание этого, необходимо предусмотреть способ для автоматического определения размеров массивов, после чего вам уже не придется вспоминать, сколько элементов содержит массив.

Такой способ существует, и он основывается на простой идее о том, что в конце каждой строки помещается специальный символ, свидетельствующий об окончании строки. При таком способе задания строки уже можно легко найти конец этой строки и любая функция может произвести подсчет количества символов, содержащихся в строке. Разрабатывая все свои функции с учетом того, что в конце каждой строки стоит специальный символ, вы можете исключить передачу дополнительного числа, задающего размер массива.

В языке программирования C таким символом, сигнализирующим об окончании строки, является нулевой символ, который записывается как `'\0'`. Поэтому в утверждении

```
const char word [] = { 'H', 'e', 'l', 'l', 'o', '!', '\0' };
```

задается символьный массив с именем `word`, который содержит семь символов, последний из которых является нулевым символом. (Вспомните, что символ обратная черта `'\'` в языке программирования C является специальным символом и не рассматривается как значащий символ. Поэтому запись `'\0'` представляет всего один символ.) Массив `word` изображен на рис. 10.2.

word[0]	'H'
word[1]	'e'
word[2]	'l'
word[3]	'l'
word[4]	'o'
word[5]	'!'
word[6]	'\0'

Рис. 10.2. Массив `word` с заключающим нулевым символом

Для демонстрации того, как можно использовать символьные массивы переменной длины, сделаем следующее. Напишем функцию, которая подсчитывает количество символов в символьной строке, как показано в листинге 10.2. Вызовем функцию и передадим ей в качестве аргумента символьный массив, который заканчивается нулевым символом. В функции будет произведен подсчет количества символов и это значение будет возвращено в вызвавшую подпрограмму. Определение числа символов производится путем подсчета количества символов до нулевого символа, который не включается в результат. Поэтому при вызове функции

```
stringLength (characterString)
```

будет возвращено число 3, если переменная `characterString` была объявлена следующим образом.

```
char characterString[] = { 'c', 'a', 't', '\0' };
```

Листинг 10.2. Подсчет количества символов в строке

```
// Функция подсчета количества символов в строке
#include <stdio.h>
int stringLength (const char string[])
{
    int count = 0;
    while ( string[count] != '\0' )
        ++count;
    return count;
}

int main (void)
{
    int stringLength (const char string[]);
    const char word1[] = { 'a', 's', 't', 'e', 'r', '\0' };
    const char word2[] = { 'a', 't', '\0' };
    const char word3[] = { 'a', 'w', 'e', '\0' };
    printf ("%i %i %i\n", stringLength (word1),
        stringLength (word2), stringLength (word3));
    return 0;
}
```

Листинг 10.2. Вывод

```
5 2 3
```

В функции `stringLength` параметр объявлен как символьный массив с ключевым словом `const`, поскольку в массиве не должно производиться никаких изменений, а только подсчет количества символов. Внутри функции `stringLength` объявлена переменная `count`, значение которой устанавливается в 0 при каждом вызове функции. Также в теле функции используется цикл `while` для последовательного подсчета всех символов, пока не встретится нулевой символ. После достижения нулевого символа, свидетельствующего о том, что достигнут конец строки, происходит выход из цикла `while` и возвращается значение переменной `count`. Это значение и соответствует количеству всех символов, за исключением нулевого символа.

Для проверки правильности работы функции лучше всего использовать небольшие символьные массивы, с помощью которых можно легко подсчитать общее количество символов. В подпрограмме `main` для проверки использованы три символьных массива: `word1`, `word2` и `word3`. С помощью функции `printf` отображаются результаты подсчета количества символов функцией `strlen` для каждого из этих массивов.

Инициализация и отображение символьных массивов

Сейчас целесообразно вернуться к функции `concat`, которая была разработана в листинге 10.1, и переписать ее, с тем чтобы она могла работать с символьными строками переменной длины. Очевидно, что в функции должны быть сделаны некоторые изменения, так как, например, больше нет необходимости передавать в функцию размеры массивов, т.е. надо передавать только три аргумента: это два символьных массива, которые должны быть объединены, и еще один массив, в который будет помещен результат объединения.

Перед тем как начать разрабатывать эту программу, вы должны познакомиться с двумя важными возможностями, которые язык программирования C предлагает для работы с символьными строками.

Первая возможность — это инициализация символьных массивов. Язык C разрешает инициализацию символьного массива одной константной символьной строкой, а не списком отдельных символов. Поэтому утверждение

```
char word[] = { "Hello!" };
```

задает символьный массив `word`, который инициализируется последовательностью символов `'H', 'e', 'l', 'l', 'o', '!', '\0'` соответственно. Также можно не ставить фигурные скобки при инициализации массива, как сделано в следующем утверждении.

```
char word[] = "Hello!";
```

Это утверждение вполне корректно. Следующее утверждение полностью эквивалентно предыдущему.

```
char word[] = { 'H', 'e', 'l', 'l', 'o', '!', '\0' };
```

Если вы явно указываете размер массива, убедитесь, что вы выделяете достаточно места для размещения строки и конечного нулевого символа. Поэтому в утверждении

```
char word[7] = { "Hello!" };
```

компилятор выделит достаточно места и разместит нулевой символ. Но если написать

```
char word[6] = { "Hello!" };
```

то компилятор не сможет разместить в конце строки нулевой символ, т.к. для него не выделено места

Запомните, что, в любом случае, ко всем символьным константам в языке программирования C будет автоматически добавляться нулевой символ. Именно это используют функции наподобие `printf` для определения конца строки. Поэтому при вызове функции

```
printf ("Programming in C is fun.\n");
```

нулевой символ будет автоматически помещен в конец символьной константы, после символа перехода на новую строку. Этого вполне достаточно для того, чтобы функция `printf` могла обнаружить окончание строки.

Вторая из двух упоминаемых возможностей касается удобства вывода на экран символьных строк. Для этого используется специальный символ форматирования '%s' в строке форматирования функции printf, с помощью которого можно отобразить массив символов, оканчивающихся нулевым символом. Следовательно, если символьный массив word оканчивается нулевым символом, то вызов функции

```
printf ("%s\n", word);
```

можно использовать для отображения всего содержимого массива word. В этом случае при наличии символа форматирования %s функция printf предполагает, что соответствующий аргумент является символьной строкой с нулевым окончанием.

Обе эти возможности использованы в подпрограмме main для программы из листинга 10.3, в которой описывается улучшенная функция concat. Поскольку в функцию уже не передается количество символов каждого массива, то длина массивов вычисляется в функции, для чего используется нулевой символ. Также при копировании массива str1 в массив result, необходимо сделать так, чтобы нулевой символ не был скопирован, т.к. конец суммарной строки должен быть после второй строки. Но при этом нет необходимости ставить нулевой символ после копирования второй строки, его можно скопировать вместе со строкой. И именно этот нулевой символ будет сигнализировать об окончании суммарной строки.

Листинг 10.3. Объединение символьных строк

```
#include <stdio.h>
int main (void)
{
    void concat (char result[], const char str1[], const char str2[]);
    const char s1[] = { "Test " };
    const char s2[] = { "works." };
    char s3[20];

    concat (s3, s1, s2);
    printf ("%s\n", s3);

    return 0;
}

// Функция объединения двух символьных строк.
void concat (char result[], const char str1[], const char str2[])
{
    int i, j;
    // Копирование str1 в result.
    for ( i = 0; str1[i] != '\0'; ++i )
        result[i] = str1[i];
    // Копирование str2 в result.
    for ( j = 0; str2[j] != '\0'; ++j )
        result[i + j] = str2[j];
    // Заканчиваем объединенную строку нулевым символом.
    result [i + j] = '\0';
}
```

Листинг 10.3. Вывод

```
Test works.
```

В первом цикле `for` для функции `concat` символы, содержащиеся внутри массива `str1`, копируются в массив `result` до тех пор, пока не встретится нулевой символ. Поскольку цикл `for` прекращается в тот момент, когда будет достигнут нулевой символ, то этот символ не копируется в массив `result`.

Во втором цикле уже символы из массива `str2` копируются в массив `result` сразу же за последним символом массива `str1`, поскольку при завершении первого цикла `for` значение переменной будет равно количеству символов массива `str1`, исключая нулевой символ. Поэтому утверждение

```
result[i + j] = str2[j];
```

используется для того, чтобы скопировать символы из массива `str2` на нужные позиции массива `result`.

После окончания второго цикла, в функции `concat` производится размещение нулевого символа в конце строки. Обратите особое внимание на использование переменных `i` и `j`. Убедитесь, что вы хорошо понимаете их назначение. Очень много ошибок происходит именно из-за того, что при работе со символьными массивами неправильно используются индексы. Вспомните, что для первого символа массива используется индекс 0.

Также напомним, что если символьный массив `string` содержит `n` символов, включая нулевой символ, то выражение `string[n-1]` ссылается на последний (не нулевой) символ строки, а выражение `string[n]` ссылается на нулевой символ. Кроме того, массив `string` должен быть объявлен для хранения `n+1` символов, поскольку нулевой символ тоже входит в состав массива.

Возвращаясь к программе, отметим, что в подпрограмме `main` объявлено два массива `s1` и `s2` типа `char`, значения для которых задаются с помощью нового способа, описанного несколько выше. Массив `s3` объявлен для хранения 20 символов, что должно хватить для копирования двух символьных массивов. При этом размер массива берется с запасом, во избежание возможных проблем и подсчета точного количества символов.

Затем вызывается функция `concat` с тремя строками в качестве аргументов. `s1`, `s2`, и `s3`.

По окончании работы функции `concat`, результирующая строка будет находиться в массиве `s3`. Она отображается с использованием символа форматирования `%s`. Хотя для строки `s3` выделено место достаточное для размещения 20 символов, функция `printf` отобразит только те символы, которые расположены перед нулевым символом.

Сравнение двух символьных строк

Вы не сможете сравнить две строки, используя следующее утверждение

```
if ( string1 == string2 )
```

```
...
```

поскольку оператор равенства может использоваться только с переменными простого типа, такими как `float`, `int` или `char`, и не может применяться совместно с более сложными типами, такими как структуры или массивы.

Для определения равенства двух массивов, вы должны непосредственно сравнить содержимое двух символьных массивов, символ за символом. Если будет достигнут конец обоих массивов одновременно, и если все символы идентичны, то тогда можно считать массивы равными, иначе они не равны.

Поэтому вполне целесообразно разработать функцию, которая производит сравнение двух символьных массивов, как показано в листинге 10.4. Вы можете использовать

функцию `equalStrings` для сравнения двух символьных строк. Поскольку вас интересует только равенство двух символьных строк, функция должна возвращать булево значение `TRUE` (или не ноль), если строки идентичны, и `FALSE` (или ноль) – в противном случае. Таким образом, функция может использоваться внутри условного выражения наподобие следующего.

```
if ( equalStrings (string1, string2) )
```

```
...
```

Листинг 10.4. Сравнение двух строк

```
// Функция определения равенства двух строк.
#include <stdio.h>
#include <stdbool.h>
bool equalStrings (const char s1[], const char s2[])
{
    int i = 0;
    bool areEqual;

    while ( s1[i] == s2 [i] &&
            s1[i] != '\0' && s2[i] != '\0' )
        ++i;
    if ( s1[i] == '\0' && s2[i] == '\0' )
        areEqual = true;
    else
        areEqual = false;

    return areEqual;
}

int main (void)
{
    bool equalStrings (const char s1[], const char s2[]);
    const char stra[] = "string compare test";
    const char strb[] = "string";

    printf ("%i\n", equalStrings (stra, strb));
    printf ("%i\n", equalStrings (stra, stra));
    printf ("%i\n", equalStrings (strb, "string"));

    return 0;
}
```

Листинг 10.4. Вывод

```
0
1
1
```

Функция `equalStrings` использует цикл `while` для последовательного просмотра символьных строк `s1` и `s2`. Цикл выполняется до тех пор, пока символы равны (`s1[i]==s2[i]`) и пока они не достигнут конца одной из строк (`s1[i] != '\0' && s2[i] != '\0'`). Переменная `i`, которая используется в качестве индекса для обоих массивов, инкрементируется при каждой итерации.

Утверждение `if` выполняется после окончания цикла `while`, с тем чтобы проверить одновременное достижение конца обоими массивами `s1` и `s2`. Здесь для сравнения последних символов должно использоваться следующее утверждение.

```
if ( s1[i] == s2[i] )  
...
```

Если эти символы являются нулевыми символами, то символьные массивы равны. В этом случае переменной `areEqual` присваивается значение `TRUE`, которое возвращается в вызывающую подпрограмму. В противном случае строки не идентичны и переменная `areEqual` принимает значение `FALSE`.

В подпрограмме `main` создаются два символьных массива `stra` и `strb`, которым присваиваются начальные значения. При первом вызове функции `equalStrings`, в нее передаются эти массивы в качестве аргументов. Поскольку эти массивы не равны, функция совершенно справедливо возвращает значение `FALSE`, или `0`.

При втором вызове функции `equalStrings`, в нее дважды передается массив `stra`. Функция корректно возвращает значение `TRUE`, свидетельствующее о том, что эти массивы равны. Это следует из вывода, сделанного программой.

В третьем случае вызов функции `equalStrings` связан с более сложными проблемами. Как можно видеть, здесь производится передача константной символьной строки в функцию, где в качестве аргумента ожидается символьный массив. В главе 11 “Указатели”, этот случай будет рассмотрен более подробно.

Функция `equalStrings` сравнивает символы строки `strb` с символами строки “string” и возвращает значение `TRUE`, свидетельствующее о том, что эти строки равны.

Ввод символьных строк

На данном этапе вы уже должны знать, как отображать символьные строки с помощью символа форматирования “%s”. Но вы еще не знаете, как считывать символы с экрана (или с окна терминала). Для этого в системе предусмотрено несколько библиотечных функций. Функция `scanf` также может использовать символ форматирования “%s” для считывания символьной строки с экрана, которая заканчивается пробелом, символом табуляции или переходом на новую строку. Поэтому написав следующие утверждения

```
char string[81];  
scanf ("%s", string);
```

вы сможете считать строку, введенную в окно терминала, и сохранить ее в символьном массиве `string`. Обратите внимание, в отличие от предыдущих вызовов функции `scanf`, в этом случае нет необходимости ставить символ “&” перед именем массива, что вполне закономерно; более подробно это будет обсуждаться в главе 11.

Если произвести вызов функции `scanf` и ввести следующую строку:

```
Shawshank
```

то строка “Shawshank” будет считана с помощью функции `scanf` и сохранена в массиве `string`. Однако если вместо этой строки набрать следующий текст

```
iTunes playlist
```

то только одно слово “iTunes” будет сохранено в массиве `string`, поскольку пробел после этого слова укажет функции `scanf` на конец считывания. Но если вызвать функцию `scanf` еще раз, то будет считано второе слово “playlist” и сохранено в массиве

string, поскольку функция scanf всегда продолжает считывание после последнего считанного символа.

Функция scanf всегда автоматически дополняет считанную строку нулевым символом. Поэтому, вызывая функцию scanf для считывания следующего текста

```
abcdefghijklmnopqrstuvwxyz
```

вы получите массив string, заполненный 26-ю строчными буквами алфавита с нулевым символом в позиции string[26].

Если задать символьные массивы s1, s2 и s3 соответствующего размера, то выполнение следующего утверждения

```
scanf ("%s%s%s", s1, s2, s3);
```

для введенного текста

```
micro computer system
```

приведет к заполнению массивов s1, s2 и s3 словами "micro", "computer", "system" соответственно. Если написать следующий текст

```
system expansion
```

то будут заполнены только массивы s1 и s2 словами "system" и "expansion". При этом отсутствие третьего слова в тексте приведет к тому, что функция scanf будет ожидать дополнительного ввода от пользователя в окне терминала.

В программе из листинга 10.5 функция scanf используется для считывания трех символьный строк.

Листинг 10.5. Считывание строк с помощью функции scanf

```
// Программа демонстрирует использование функции scanf
// с символом форматирования "%s"
#include <stdio.h>
int main (void)
{
    char s1[81], s2[81], s3[81];

    printf ("Enter text:\n");
    scanf ("%s%s%s", s1, s2, s3);
    printf ("\ns1 = %s\ns2 = %s\ns3 = %s\n", s1, s2, s3);

    return 0;
}
```

Листинг 10.5. Вывод

```
Enter text:
system expansion
bus
s1 = system
s2 = expansion
s3 = bus
```

В программе функция scanf вызывается для считывания в три символьных массива: s1, s2 и s3. Поскольку первая строка содержит только два слова (по определению

для функции `scanf` слово заканчивается пробелом, символом табуляции или переходом на новую строку), то функция ожидает дополнительного ввода текста. После этого вызывается функция `printf` для проверки правильности ввода и сохранения слов "system", "expansion" и "bus" в массивах `s1`, `s2` и `s3` соответственно.

Если в предыдущей программе вы введете более 80 последовательных символов без пробела, символа табуляции или нажатия клавиши <Enter>, то функция `scanf` переполнит один из своих массивов. Это может привести к аварийному завершению программы или непредсказуемому результату. К сожалению, функция `scanf` не имеет средств для определения длины слова. Когда используется символ форматирования "%s", то при этом производится последовательное считывание и заполнение массива до тех пор, пока не встретится символ окончания слова.

Но если разместить некоторое число после символа "%" в строке форматирования, то это число будет использоваться как граничное значение для числа считываемых символов. Поэтому можно использовать вызов функции `scanf` следующим образом

```
scanf ("%80s%80s%80s", s1, s2, s3);
```

а не так, как это сделано в листинге 10.5. При этом функция будет "знать", что в массивах `s1`, `s2` и `s3` нельзя сохранять больше чем 80 символов. (Вы должны оставить место для заключительного нулевого символа, поэтому используется значение "%80s" вместо "%81s".)

Ввод отдельных символов

В стандартной библиотеке предусмотрено несколько функций для чтения и записи отдельных символов и целых слов. В различных ситуациях можно использовать наиболее подходящие из них. Функция с именем `getchar` может использоваться для считывания отдельного символа с терминала. Повторный вызов функции `getchar` приведет к считыванию следующего символа из входного потока. Когда будет достигнут конец строки, то будет считан символ перехода на новую строку — '\n'. Поэтому если в окне терминала набрать слово "abc" с последующим нажатием клавиши <Enter>, то после первого вызова функции `getchar` будет считан символ 'a', второй вызов функции `getchar` приведет к считыванию символа 'b', третий вызов функции считает символ 'c', а после четвертого вызова функции `getchar` будет возвращен символ новой строки '\n'. Пятый вызов функции заставит программу ожидать ввода очередного символа.

Вас может удивить сам факт включения в состав библиотеки функции для считывания отдельного символа, когда символ можно считать с помощью уже существующей функции `scanf`, используя символ форматирования '%c'. С помощью функции `scanf` можно корректно выполнить считывание одиночного символа, однако функция `getchar` лучше выполняет эту задачу, т.к. она не является универсальной и рассчитана на считывание только отдельного символа и поэтому не требует никаких аргументов. Функция возвращает отдельный символ, который можно присвоить переменной или использовать для решения других задач в программе.

Во многих приложениях, связанных с обработкой текста, требуется считывание целой строки текста. Эта строка считывается в определенное место памяти, называемое "буфером", для дальнейшей обработки. Если в этом случае применить функцию `scanf` с символом форматирования '%s', то идея не сработает, т.к. ввод строки будет прекращен при встрече первого же пробела. При этом ввод сразу же прекратится.

Поэтому для считывания целой строки нужно использовать функцию `gets`. Как вы уже догадались, основная задача этой функции — это считывание отдельной текстовой строки. В листинге 10.6 приведен пример использования функции `readLine`, которая

подобна функции `gets` и разработана с использованием функции `getchar`. Функция принимает единственный аргумент — символьный массив, в котором должна сохраняться считанная строка. Строка считывается с окна терминала вплоть до символа перехода к новой строке, который не включается в выходной массив.

Листинг 10.6. Считывание строк данных

```
#include <stdio.h>
int main (void)
{
    int i;
    char line[81];
    void readLine (char buffer[]);

    for ( i = 0; i < 3; ++i )
    {
        readLine (line);
        printf ("%s\n\n", line);
    }
    return 0;
}

// Функция считывает строку текста с терминала.
void readLine (char buffer[])
{
    char character;
    int i = 0;
    do
    {
        character = getchar();
        buffer[i] = character;
        ++i;
    }
    while ( character != '\n' );
    buffer[i - 1] = '\0';
}
```

Листинг 10.6. Вывод

```
This is a sample line of text.
This is a sample line of text.
Abcdefghijklmnopqrstuvwxyz
abcdefghijklmnopqrstuvwxy
runtime library routines
runtime library routines
```

Цикл `do` в функции `readLine` используется для построения копии входной строки внутри символьного массива `buffer`. Каждый символ, который возвращается функцией `getchar`, сохраняется в очередной позиции массива. Когда будет достигнут символ перехода на новую строку, что свидетельствует об окончании строки, цикл прекращается, но при этом символ перехода на новую строку записывается в массив. Затем в массив добавляется нулевой символ, который замещает переход на новую строку. Индекс с номером `i-1` указывает на правильную позицию в массиве, поскольку номер индекса был дополнительно инкрементирован перед выходом из цикла.

В подпрограмме `main` объявляется символьный массив с именем `line`, размер которого задается равным 81 символу, что вполне достаточно для сохранения строки. При этом для символов строки выделяется 80 символов (это значение принято использовать для “стандартного терминала”), и еще одна позиция для нулевого символа. Однако даже в окне терминала с длиной строки 80 или меньше символов все еще остается опасность переполнения массива, если вы продолжаете вводить символы после того, как нажали клавишу `<Enter>`. Поэтому будет правильно расширить функцию `readLine` и добавить еще один параметр — размер буфера. В этом случае функция может контролировать заполнение буфера и не превышать указанный предел.

Затем в программе вводится цикл `for`, который используется для того, чтобы вызвать функцию `readLine` три раза. При каждом вызове функции очередная строка текста считывается с экрана терминала. Каждая считанная строка повторяется на экране терминала для того, чтобы удостовериться в правильности работы функции. После того как будет отображена третья строка текста, выполнение программы из листинга 10.6 завершается.

В следующем примере (листинг 10.7) рассматривается приложение, которое можно использовать в практических целях — подсчета количества слов в выделенном тексте. Разработаем функцию с именем `countWords`, которая принимает в качестве аргумента символьную строку и возвращает число слов, содержащихся в строке. Для упрощения задачи предположим, что слово является последовательностью одного или нескольких букв алфавита. Функция будет сканировать строку до появления первой буквы и затем просматривать последовательность букв до появления символа, не являющегося буквой. Затем функция продолжает сканировать строку до появления очередной буквы, являющейся началом очередного слова.

Листинг 10.7. Подсчет количества слов

```
// Function to determine if a character is alphabetic
#include <stdio.h>
#include <stdbool.h>
bool alphabetic (const char c)
{
    if ( (c >= 'a' && c <= 'z') || (c >= 'A' && c <= 'Z') )
        return true;
    else
        return false;
}

/* Function to count the number of words in a string */
int countWords (const char string[])
{
    int i, wordCount = 0;
    bool lookingForWord = true, alphabetic (const char c);
    for ( i = 0; string[i] != '\0'; ++i )
        if ( alphabetic(string[i]) )
        {
            if ( lookingForWord )
            {
                ++wordCount;
                lookingForWord = false;
            }
        }
    else
        lookingForWord = true;
```

```

    return wordCount;
}

int main (void)
{
    const char text1[] = "Well, here goes.";
    const char text2[] = "And here we go... again.";
    int countWords (const char string[]);

    printf ("%s - words = %i\n", text1, countWords (text1));
    printf ("%s - words = %i\n", text2, countWords (text2));

    return 0;
}

```

Листинг 10.7. Вывод

```

Well, here goes. - words = 3
And here we go... again. - words = 5

```

Функция `alphabetic` достаточно проста и понятна, она проверяет принадлежность символа к строчной или заглавной букве. Если это буква, функция возвращает значение `true`, в противном случае возвращается значение `false`.

Функция `countWords` несколько сложнее. Целочисленная переменная `i` используется в качестве индексного значения для последовательности символов строки. Целочисленная переменная `lookingForWord` используется в качестве флага, применяемого для индикации того, что происходит процесс поиска начала нового слова. Понятно, что в начале выполнения функции вы находитесь в режиме поиска начала нового слова, поэтому флаг устанавливается в значение `true`. Локальная переменная `wordCount` используется для подсчета количества слов в символьной строке.

Функция `alphabetic` вызывается для каждого символа строки для определения того, является ли он буквой. Если символ является буквой, то производится проверка флага `lookingForWord` для определения того, что производится поиск нового слова. Если это так, то значение `wordCount` инкрементируется на 1, а для флага `lookingForWord` устанавливается значение `false`, свидетельствующее о том, что начало слова найдено и его поиск больше не производится.

Если символ является буквой и флаг `lookingForWord` имеет значение `false`, то это означает, что в настоящий момент считывается буква, находящаяся внутри слова. В этом случае цикл `for` продолжается, т.е. считывается следующий символ строки.

Если символ не является буквой, то это означает, что вы достигли конца слова или что вы еще не нашли начала слова — для флага `lookingForWord` устанавливается значение `true`, даже если оно уже было `true`.

Когда все символы внутри символьной строки будут проверены, функция возвращает значение переменной `wordCount`, представляющее количество слов, определенное в символьной строке.

Для лучшего понимания алгоритма работы полезно составить таблицу различных значений переменных в функции `countWords`. Такие значения приведены в таблице 10.1, начиная с первого вызова функции `countWords` в приведенном примере программы. Первая строка таблицы 10.1 показывает начальные значения переменных `wordCount` и `lookingForWord` перед переходом к циклу `for`. Последующие строки отображают значения указанных переменных для каждой очередной итерации цикла `for`.

Поэтому во второй строке таблицы показано, что значение переменной `wordCount` будет установлено в 1, а значение флага `lookingForWord` станет равно `false` (0) после первого прохода цикла (после считывания символа 'w'). Последняя строка таблицы отображает значения переменных, которые будут получены при достижении конца строки. Вы должны внимательно изучить эту таблицу, сопоставляя ее с логикой работы функции `countWords`. Лишь убедившись в том, что вы хорошо понимаете работу функции, вы сможете вносить в нее необходимые изменения.

Таблица 10.1. Работа функции `countWords`

i	string[i]	wordCount	lookingForWord
		0	TRUE
0	'w'	1	FALSE
1	'e'	1	FALSE
2	'l'	1	FALSE
3	'l'	1	FALSE
4	','	1	TRUE
5	' '	1	TRUE
6	'h'	2	FALSE
7	'e'	2	FALSE
8	'r'	2	FALSE
9	'e'	2	FALSE
10	' '	2	TRUE
11	'g'	3	FALSE
12	'o'	3	FALSE
13	'e'	3	FALSE
14	's'	3	FALSE
15	'.'	3	TRUE
16	'\0'	3	TRUE

Строка Null

Теперь рассмотрим небольшой практический пример использования функции `countWords`. Функция `readLine` будет доработана таким образом, чтобы позволить пользователю вводить несколько строк текста с окна терминала, после чего программа подсчитает количество слов в тексте и отобразит результат.

Для того чтобы программа стала более универсальной, количество вводимых строк текста не будет ограничиваться или задаваться отдельно. Поэтому необходимо каким-то образом дать знать программе, что ввод текста окончен. Самым приемлемым способом в данном случае будет нажатие клавиши <Enter> после того, как ввод текста закончен.

Когда функция `readLine` вызывается для считывания такой строки, то в ней учитываются символы перехода на новую строку, и в результате после дополнительного нажатия клавиши <Enter> в буфер записывается нулевой символ. При этом можно будет проверять наличие строки, не содержащей ни одного символа. Это будет сигнализировать об окончании текста.

Символьная строка, которая не содержит никаких символов, кроме нулевого, имеет специальное название в языке программирования С — нулевая строка. Если вспомнить о всех функциях этой главы, то можно отметить, что использование нулевой строки хорошо согласуется с работой всех функций. Функция `strlen` корректно возвращает 0 как размер нулевой строки, функция `concat` также использует нулевую строку в конце, даже разработанная функция `equalStrings` будет работать правильно, если обе строки будут нулевыми (в этом случае функция корректно посчитает эти строки равными).

Всегда помните, что нулевая строка все же имеет символ, хотя и нулевой. Иногда возникает необходимость сделать нулевой символьную строку. В языке программирования С нулевая строка задается парой смежных двойных кавычек. Поэтому в утверждении

```
char buffer[100] = "";
```

будет объявлен символьный массив с именем `buffer`, которому в качестве значения будет присвоена нулевая строка. Обратите внимание, что символьная строка "" — это не то же самое, что символьная строка " ", поскольку во втором случае в строке содержится символ пробела. Для того чтобы можно было их различать, используйте эти строки с функцией `equalStrings` и посмотрите результат.

В программе листинга 10.8 используются функции `readLine`, `alphabetic` и `countWords` из предыдущей программы. В листинге они не приводятся из-за ограничений по объему.

Листинг 10.8. Подсчет слов во фрагменте текста

```
#include <stdio.h>
#include <stdbool.h>
/***** Вставить функцию alphabetic сюда *****/
/***** Вставить функцию readLine сюда *****/
/***** Вставить функцию countWords сюда *****/
int main (void)
{
    char text[81];
    int totalWords = 0;
    int countWords (const char string[]);
    void readLine (char buffer[]);
    bool endOfText = false;

    printf ("Type in your text.\n");
    printf ("When you are done, press 'RETURN'.\n\n");
    while ( ! endOfText )
    {
        readLine (text);
        if ( text[0] == '\0' )
            endOfText = true;
        else
            totalWords += countWords (text);
    }
    printf ("\nThere are %i words in the above text.\n", totalWords);

    return 0;
}
```

Листинг 10.8. Вывод

```
Type in your text.  
When you are done, press 'RETURN'.  
Wendy glanced up at the ceiling where the mound of lasagna loomed  
like a mottled mountain range. Within seconds, she was crowned with  
ricotta ringlets and a tomato sauce tiara. Bits of beef formed meaty  
moles on her forehead. After the second thud, her culinary coronation  
was complete.  
Enter  
There are 48 words in the above text.
```

В строке, помеченной как “Enter”, напоминает о нажатии клавиши <Enter>. Переменная `endOfText` используется как флаг для указания на то, что достигнут конец текста. Цикл `while` выполняется до тех пор, пока флаг не примет значение `false`. Внутри цикла производится вызов функции `readLine` для считывания строк текста. В утверждении `if` производится проверка входной строки, которая сохранена в массиве `text` в целях проверки того, что нажата клавиша <Enter>. Если это так, то буфер содержит нулевую строку. В этом случае флаг `endOfText` устанавливается в состояние `true`, свидетельствующее о том, что достигнут конец ввода.

Если буфер не содержит никакого текста, вызывается функция `countWords` для подсчета количества слов в массиве `text`. Это значение возвращается и суммируется со значением переменной `totalWords`, которая содержит нарастающее количество слов со всех считанных строк вводимого текста.

После завершения цикла `while`, программа отображает значение переменной `totalWords` с небольшим поясняющим текстом. Но и эта программа не очень облегчает работу, т.к. приходится вводить весь текст вручную. Но, как вы увидите в главе 16, “Операции ввода и вывода в языке C”, эта же самая программа, например, может использоваться для подсчета количества слов, содержащихся в файле на диске.

Поэтому каждый, кто использует компьютеры для работы с текстами, найдет эту программу чрезвычайно полезной, т.к. с ее помощью можно быстро подсчитать количество слов в тексте. При этом предполагается, что используется обычный текстовый файл, а не форматированный текст Microsoft Word.

Переходные символы

Как отмечалось ранее, символ обратной черты имеет специальное назначение и позволяет, например, формировать символ новой строки или нулевой символ. Но его применение гораздо шире. Аналогично комбинации символа обратной черты и символа `n`, которая используется для перехода на новую строку при печати, с символом обратной черты можно использовать и другие символы для выполнения определенных задач.

Любую комбинацию с символом обратной черты часто называют *переходным символом*. Соответствующие комбинации приведены в таблице 10.2.

Table 10.2. Переходные символы

Переходной символ	Название
<code>\a</code>	Предупреждающий сигнал
<code>\b</code>	Возврат
<code>\f</code>	Подача страницы
<code>\n</code>	Разделитель строк
<code>\r</code>	Возврат каретки
<code>\t</code>	Горизонтальная табуляция
<code>\v</code>	Вертикальная табуляция
<code>\\</code>	Обратная черта
<code>\"</code>	Двойные кавычки
<code>\'</code>	Одинарные кавычки
<code>\?</code>	Знак вопроса
<code>\nnn</code>	Восьмеричное значение nnn
<code>\unnnn</code>	Универсальный символ
<code>\Unnnnnnnn</code>	Универсальный символ
<code>\xnn</code>	Шестнадцатеричное значение nn

Первые семь символов, перечисленные в таблице 10.2, выполняют указанные функции в большинстве устройств вывода, когда встречаются в тексте. Предупреждающий сигнал звучит, как звонок в большинстве терминалов. Поэтому за вызовом функции `printf`

```
printf ("\aSYSTEM SHUT DOWN IN 5 MINUTES!!\n");
```

последует звуковой сигнал с последующим выводом строки. Включение переходного символа `'\b'` внутри символьной строки приведет к возврату курсора на один символ назад при условии, что этот режим поддерживается окном терминала. Аналогично, при вызове функции

```
printf ("%i\t%i\t%i\n", a, b, c);
```

сначала отобразится значение переменной `a`, затем будет выполнен символ табуляции (по умолчанию обычно восемь пробелов), после чего отобразится символ `b` и еще восемь пробелов, затем будет выведен символ `c` и сделан переход на новую строку. Символ горизонтальной табуляции обычно используется для выравнивания выводимых данных по колонкам.

Для включения обратной черты внутри символьной строки, необходимо исповедовать две обратные черты, поэтому вызов функции `printf` в следующем виде

```
printf ("\\t is the horizontal tab character.\n");
```

приведет к отображению следующего текста.

```
\t is the horizontal tab character.
```

Обратите внимание, что в начале строки стоят две обратные черты и символ `"t"`, но в этом случае не выполняется табуляция.

Для включения двойных кавычек в символьную строку, они должны стоять за обратной чертой. Поэтому вызов функции `printf` в следующем виде

```
printf ("\"Hello,\" he said.\n");
```

отобразит на экране следующее.

```
"Hello," he said.
```

Для того чтобы присвоить символ одиночной кавычки символьной переменной, перед одинарной кавычкой должна находиться обратная черта. Если переменная объявлена как переменная типа `char`, то в следующем утверждении

```
c = '\';
```

этой переменной будет присвоено значение в виде символа одиночной кавычки.

Символ обратной черты с последующим символом “?” используется для представления знака вопроса. Это иногда необходимо при работе с наборами символов не ASCII-формата. Подробнее об этом говорится в приложении А, “Краткое изложение языка C”.

С помощью последних четырех элементов таблицы 10.2 можно включить любой символ в символьную строку. В переходном символе “\nnn”, символы `nnn` представляют три восьмеричных числа. В переходном символе “\xnn”, `nn` — шестнадцатеричное число. Эти числа представляют цифровой код символа. При этом можно использовать символы, которые недоступны для ввода с клавиатуры или которые нельзя представить в символьной строке. Например, для того чтобы включить ASCII-символ, который имеет восьмеричное значение 33, вы должны записать последовательности `\033` или `\x1b` внутри соответствующей строки.

Нулевой символ, описанный в предыдущем параграфе, отражает специальный случай переходного символа. Он представляет символ, цифровое значение которого равно нулю. На самом деле, поскольку значение нулевого символа равно 0, это часто используется программистами для контроля циклов, в которых обрабатываются строки переменной длины. Например, цикл для подсчета длины символьной строки в функции `stringLength` из листинга 10.2 может быть записан следующим образом.

```
while ( string[count] )  
++count;
```

Значение `string[count]` не будет равняться нулю до тех пор, пока не будет достигнут конец строки, после чего произойдет выход из цикла `while`.

Необходимо еще раз напомнить, что все переходные символы рассматриваются как один символ. Поэтому символьная строка “\033\“Hello\“\n” содержит всего девять символов (не включая нулевой символ): символ ‘\033’, символ двойных кавычек ‘\’, пять символов слова `Hello`, опять символ двойных кавычек и символ разделения строк. Попробуйте проверить эту строку с помощью функции `stringLength` и вы убедитесь, что в строке действительно девять символов.

Универсальные символы формируются с помощью символа `\u` с последующим четырехзначным шестнадцатеричным числом, или символом `\U` с последующим восьмизначным шестнадцатеричным числом. Такие символы используются для представления символов из расширенного набора символов, т.е. для таких символов, которые для своего представления требуют более чем восемь стандартных битов. Универсальные символы используются для формирования идентификаторов из расширенного набора данных, или из 16- и 32-разрядных символов внутри строк из расширенных символов и строковых констант. Подробнее об этом рассказывается в приложении А.

Более подробно о строковых константах

Если поместить символ обратной черты в самом конце строки и непосредственно за ним ввести символ возврата каретки, то компилятор проигнорирует конец строки. Такой способ обычно используется для продолжения длинных символьных строк на следующую строку экрана и, как вы увидите в главе 13, “Препроцессор”, для размещения макроопределений на нескольких строках.

Без символа продолжения строки компилятор языка C сгенерирует ошибку и выведет сообщение о том, что вы пытаетесь инициализировать символьную строку на нескольких строках экрана.

```
char letters[] =
{ "abcdefghijklmnopqrstuvwxyz
ABCDEFGHIJKLMNOPQRSTUVWXYZ" };
```

Размещая символ обратной черты в конце каждой строки, символьную константу можно записать на нескольких строках.

```
char letters[] =
{ "abcdefghijklmnopqrstuvwxyz\
ABCDEFGHIJKLMNOPQRSTUVWXYZ" };
```

Необходимо продолжать вводить символы следующей строки непосредственно с начала строки, иначе пробелы в начале строки будут сохранены в массиве. В предыдущем примере результатом объявления массива и инициализации его элементов будет следующая символьная строка.

```
"abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ"
```

Другим способом создания длинных символьных строк будет разделение длинной строки на несколько смежных строк. Смежные строки являются строковыми константами, разделенными одним или несколькими пробелами, символами табуляции или разделителями строк. Компилятор автоматически объединяет смежные строки. Поэтому следующее стро:

```
"one" "two" "three"
```

синтаксически эквивалентно написанию одной строки:

```
"onetwothree"
```

Поэтому массив `letters` будет содержать все символы алфавита, если написать следующее.

```
char letters[] =
{ "abcdefghijklmnopqrstuvwxyz"
  "ABCDEFGHIJKLMNOPQRSTUVWXYZ" };
```

Наконец, если написать следующие три утверждения `printf`

```
printf ("Programming in C is fun\n");
printf ("Programming" " in C is fun\n");
printf ("Programming" " in C" " is fun\n");
```

то функцию всегда будет передан один аргумент, т.к. компилятор объединит все строки во втором и третьем случаях.

Символьные строки, структуры и массивы

Можно комбинировать базовые элементы языка программирования C для создания очень мощных программных конструкций. Причем это можно сделать различными способами. В главе 9, “Структуры”, например, вы могли видеть, как легко, например, создать массив структур. В листинге 10.9 можно посмотреть еще более сложную конструкцию, представляющую массив структур в комбинации с символьными строками переменной длины.

Предположим, вы хотите написать компьютерную программу, которую можно использовать в качестве словаря. Вы просто будете вводить неизвестное слово и программа автоматически будет находить и отображать его описание.

При этом следует узнать, что одной из первых проблем, которые у вас возникнут, будет представление слова и его описания в компьютере. Вполне очевидно, что поскольку слово и его описание логически взаимосвязаны, то следует использовать структурный тип. Например, вы можете задать структуру с именем `entry` для хранения слова и его описания.

```
struct entry
{
    char word[15];
    char definition[50];
};
```

В этом описании структуры задается достаточно места для 14-символьного слова (помните о нулевом символе) и 49-символьного описания слова. Ниже приводится объявление переменной типа `struct entry`, которая инициализируется словом “blob” и его описанием.

```
struct entry word1 = { "blob", "an amorphous mass" };
```

Поскольку в словаре вы будете использовать не одно слово, то необходимо создать массив структур, как показано ниже.

```
struct entry dictionary[100];
```

Этот массив рассчитан на хранение всего 100 слов. Очевидно, это недостаточно для того, чтобы разместить в нем все английские слова, которых насчитывается по крайней мере не меньше 100000. Для такого случая, вероятнее всего, надо использовать другой подход и хранить данные массива не в памяти, а на жестком диске.

Решив использовать структуры для своего словаря, необходимо подумать и о его организации. Большинство словарей используют алфавитный порядок. Имеет смысл организовать ваш словарь таким же образом. Несколько позже мы еще вернемся к этому вопросу.

А сейчас необходимо подумать о разработке самой программы. Сначала надо описать функцию, которая будет производить поиск слова в словаре. Если слово найдено, функция должна вернуть порядковый номер слова в словаре, в противном случае функция должна вернуть значение -1, свидетельствующее о том, что слово не найдено. Итак, типичный вызов функции, которую назовем `lookup`, можно представить следующим образом.

```
entry = lookup (dictionary, word, entries);
```

В этом случае функция `lookup` будет производить поиск слова `word` в словаре, имя которого `dictionary`. Третий аргумент `entries` будет содержать порядковый номер

слова после завершения поиска или содержать значение -1, свидетельствующее о том, что слово не найдено.

В программе из листинга 10.9 в функции lookup используется функция equalStrings, которая описана в листинге 10.4, для поиска соответствия заданного слова одному из слов в словаре.

Листинг 10.9. Использование программы поиска в словаре

```
// Программа использует поиск слов в словаре.
#include <stdio.h>
#include <stdbool.h>
struct entry
{
    char word[15];
    char definition[50];
};

/***** Вставить функцию equalStrings сюда *****/
// Функция для поиска слова в словаре.
int lookup (const struct entry dictionary[], const char search[],
const int entries)
{
    int i;
    bool equalStrings (const char s1[], const char s2[]);
    for ( i = 0; i < entries; ++i )
        if ( equalStrings (search, dictionary[i].word) )
            return i;
    return -1;
}

int main (void)
{
    const struct entry dictionary[100] =
        { { "aardvark", "a burrowing African mammal" },
          { "abyss", "a bottomless pit" },
          { "acumen", "mentally sharp; keen" },
          { "addle", "to become confused" },
          { "aerie", "a high nest" },
          { "affix", "to append; attach" },
          { "agar", "a jelly made from seaweed" },
          { "ahoy", "a nautical call of greeting" },
          { "aigrette", "an ornamental cluster of feathers" },
          { "ajar", "partially opened" } };
    char word[10];
    int entries = 10;
    int entry;

    int lookup (const struct entry dictionary[], const char search[],
        const int entries);
    printf ("Enter word: ");
    scanf ("%14s", word);
    entry = lookup (dictionary, word, entries);
    if ( entry != -1 )
        printf ("%s\n", dictionary[entry].definition);
    else
```



```
printf ("Sorry, the word %s is not in my dictionary.\n", word);  
return 0;  
}
```

Листинг 10.9. Вывод

```
Enter word: agar  
a jelly made from seaweed
```

Листинг 10.9. Вывод (Повторение)

```
Enter word: accede  
Sorry, the word accede is not in my dictionary.
```

Функция `lookup` последовательно просматривает все входящие в словарь слова. Для каждого слова функция `lookup` вызывает функцию `equalStrings` для определения равенства символьной строки `search` очередному слову из словаря. Если равенство найдено, функция возвращает порядковый номер слова, взятого из словаря. При этом немедленно производится выход из функции, несмотря на то, что остальные слова из словаря еще не просмотрены.

Если функция `lookup` просмотрит все слова из словаря и совпадения не будет найдено, то утверждение `return` после цикла `for` возвратит в вызывающую программу "не найдено", о чем будет свидетельствовать значение `-1`.

Улучшенный метод поиска

Метод, используемый для поиска нужного слова из словаря в функции `lookup`, достаточно прямолинеен — функция просто последовательно просматривает все слова словаря, пока не будет достигнут конец. Для небольших словарей, подобных тому, который был использован в программе, это вполне приемлемо. Однако, если вы начнете работать с большими словарями, содержащими сотни или тысячи слов, этот метод уже не подходит, т.к. на поиск отдельного слова может быть затрачено достаточно много времени, особенно если это слово находится ближе к концу словаря.

Чаще всего на поиск затрагивается довольно много времени, хотя в некоторых случаях поиск может быть выполнен и за доли секунды. Одним из основных критериев для программ поиска является среднее время поиска. Поскольку процесс поиска является одним из наиболее часто используемых в программах, специалисты уделили особое внимание разработке эффективных алгоритмов поиска (как и процессу сортировки).

Используя тот факт, что слова в словаре расположены в алфавитном порядке, можно разработать более эффективную функцию `lookup`. Во-первых, можно оптимизировать функцию за счет исключения поиска для тех слов, которых в словаре нет. Функцию `lookup` можно также сделать достаточно "интеллектуальной", с тем чтобы она прекращала поиск тогда, когда поймет, что слова в словаре нет. Например, если для поиска в словаре листинга 10.9 выбрано слово "active" то после того, как будет достигнуто слово "асимен", поиск можно прекращать, т.к. слово "active" должно быть расположено перед словом "асимен".

Такая оптимизация позволит сократить время поиска в несколько раз, но только в тех случаях, когда нужного слова в словаре нет. Но хотелось бы иметь алгоритм, который сокращает время поиска не только в отдельных случаях. Такой алгоритм существует и он называется двоичным поиском..

Стратегия, которая используется при двоичном поиске, довольно простая и легкая для понимания. Для демонстрации работы этого алгоритма, рассмотрим аналогичную ситуацию с игрой на угадывание.

Предположим, я задумал некоторое число в диапазоне от 1 до 99 и прошу вас определить это число при наименьшем количестве угадываний. После каждого предположения, которое вы сделаете, я могу ответить вам, больше ли задуманного числа предложенное вами, или меньше. После нескольких попыток, вы, вероятнее всего, поймете, что наилучшим способом сузить диапазон поиска будет выбор половинного значения диапазона. Например, если в качестве первого предположения вы назовете число 50, то искомое число может находиться или в верхнем диапазоне, или в нижнем, а может и соответствовать задуманному числу. В наиболее общем случае мой ответ “больше” или “меньше”, сузит диапазон поиска наполовину — или это будут числа от 1 до 49, или от 51 до 99.

Далее необходимо продолжать процесс деления на два в выбранном диапазоне. Поэтому, если моим ответом будет “больше”, вы будете выбирать половинку в диапазоне от 51 до 99, т.е. 75. Этот процесс должен продолжаться до тех пор, пока не будет угадано число. В среднем такой процесс поиска занимает меньше всего времени по сравнению с другими алгоритмами поиска.

В предыдущем примере довольно точно описана последовательность поиска для двоичного алгоритма поиска. Ниже сделано формальное описание этого алгоритма. В нем производится поиск элемента x внутри массива M , который содержит n элементов. При этом предполагается, что массив M отсортирован в возрастающем порядке.

Алгоритм двоичного поиска

- Шаг 1.** Установить low в 0, $high$ в $n-1$.
- Шаг 2.** Если $low > high$, x не находится в M и выполнение прекращается.
- Шаг 3.** Установить mid в $(low+high)/2$.
- Шаг 4.** Если $M[mid] < x$, установить low в $mid+1$ и перейти к шагу 2.
- Шаг 5.** Если $M[mid] > x$, установить $high$ в $mid-1$ и перейти к шагу 2.
- Шаг 6.** $M[mid]$ равно x и выполнение прекращается.

Деление, выполняемое на шаге 3, является целочисленным делением, поэтому если low равно 0, а $high$ равно 49, то значение mid будет равно 24.

Теперь, когда вы уже познакомились с алгоритмом двоичного поиска, вы можете переписать функцию `lookup` таким образом, чтобы использовать новую стратегию поиска. Поскольку при двоичном поиске вы должны не только сравнивать строки, но и определять их взаимное положение (т.е. одна строка больше другой, меньше другой или они равны), вы должны будете заменить функцию `equalStrings` другой функцией, которая определяет взаимное положение строк. Назовите функцию `compareStrings` и заставьте ее возвращать значение -1, если первая строка лексикографически меньше второй, значение 0, если строки равны; и значение 1, если вторая строка лексикографически больше первой. Поэтому вызов функции

```
compareStrings ("alpha", "altered")
```

возвратит значение -1, поскольку первая строка лексикографически меньше второй строки (это означает, что первая строка находится ближе к началу словаря, чем вторая). А вызов функции

```
compareStrings ("zioty", "yucca");
```

возвратит значение 1, поскольку "zioty" лексикографически больше, чем "yucca".

Новая функция сравнения строк `compareStrings` представлена в программе из листинга 10.10. Функция `lookup` в этом варианте использует двоичный метод поиска в словаре. Подпрограмма `main` осталась неизменной, т.е. такой же, как и в предыдущей программе.

Листинг 10.10. Модификация программы поиска в словаре с помощью двоичного поиска

```
// Программа поиска в словаре.
#include <stdio.h>
struct entry
{
    char word[15];
    char definition[50];
};
// Функция сравнения двух символьных строк.
int compareStrings (const char s1[], const char s2[])
{
    int i = 0, answer;

    while ( s1[i] == s2[i] && s1[i] != '\0' && s2[i] != '\0' )
        ++i;
    if ( s1[i] < s2[i] )
        answer = -1;    /* s1 < s2 */
    else if ( s1[i] == s2[i] )
        answer = 0;    /* s1 == s2 */
    else
        answer = 1;    /* s1 > s2 */

    return answer;
}

// Функция поиска слова в словаре.
int lookup (const struct entry dictionary[], const char search[],
const int entries)
{
    int low = 0;
    int high = entries - 1;
    int mid, result;
    int compareStrings (const char s1[], const char s2[]);

    while ( low <= high )
    {
        mid = (low + high) / 2;
        result = compareStrings (dictionary[mid].word, search);
        if ( result == -1 )
            low = mid + 1;
        else if ( result == 1 )
            high = mid - 1;
        else
            return mid; /* found it */
    }
}
```

```

    return -1; /* not found */
}

int main (void)
{
    const struct entry dictionary[100] =
        { { "aardvark", "a burrowing African mammal" },
          { "abyss", "a bottomless pit" },
          { "acumen", "mentally sharp; keen" },
          { "addle", "to become confused" },
          { "aerie", "a high nest" },
          { "affix", "to append; attach" },
          { "agar", "a jelly made from seaweed" },
          { "ahoy", "a nautical call of greeting" },
          { "aigrette", "an ornamental cluster of feathers" },
          { "ajar", "partially opened" } };
    int entries = 10;
    char word[15];
    int entry;

    int lookup (const struct entry dictionary[], const char search[],
                const int entries);
    printf ("Enter word: ");
    scanf ("%14s", word);
    entry = lookup (dictionary, word, entries);
    if ( entry != -1 )
        printf ("%s\n", dictionary[entry].definition);
    else
        printf ("Sorry, the word %s is not in my dictionary.\n", word);

    return 0;
}

```

Листинг 10.10. Вывод

```

Enter word: aigrette
an ornamental cluster of feathers

```

Листинг 10.10. Вывод (Повторение)

```

Enter word: acerb
Sorry, that word is not in my dictionary.

```

Функция `compareStrings` выполняет в цикле `while` задачи, аналогичные тем, что выполняются функцией `equalStrings`. Когда выполняется цикл `while`, функция анализирует взаимное расположение строк и возвращает соответствующее значение. Если `s1[i]` меньше чем `s2[i]`, `s1` лексикографически будет меньше чем `s2`. В этом случае возвращается значение `-1`. Если `s1[i]` равно `s2[i]`, то строки равны и возвращается значение `0`. Если ни одно из этих условий не выполняется, то строка `s1` лексикографически будет больше чем `s2` и в этом случае возвращается значение `1`.

В функции объявлены переменные `low` и `high` типа `int` и им присваиваются начальные значения для начала двоичного поиска. Цикл `while` выполняется до тех пор, пока значение переменной `low` не превысит значение переменной `high`. Внутри цикла

рассчитывается значение переменной `mid` путем сложения значений переменных `low` и `high` и деления результата на 2. Затем вызывается функция `compareStrings` со словом из словаря `dictionary[mid]` и заданным словом в качестве аргументов. Возвращаемое значение присваивается переменной `result`.

Если функция `compareStrings` возвратит значение `-1`, то это говорит о том, что слово `dictionary[mid].word` меньше заданного для поиска слова, при этом переменной `low` будет присвоено значение `mid+1`. Если функция `compareStrings` возвратит значение `1`, то это означает, что слово `dictionary[mid].search` больше, чем слово для поиска, и переменной `high` присваивается значение `mid-1`. Если возвращаемое значение не равно ни `-1`, ни `1`, то это говорит о равенстве слов и функция `lookup` возвращает значение переменной `mid`, которое указывает на порядковое положение слова в словаре.

Если значение переменной `low` превышает значение переменной `high`, заданного слова в словаре нет. В этом случае функция `lookup` возвращает значение `-1`, свидетельствующее о том, что слово не найдено.

Операции с символами

Символьные переменные и константы часто используются в логических и арифметических выражениях. Для корректного использования символов в таких ситуациях, необходимо понимать, как они обрабатываются компилятором языка C.

Когда символьные константы или переменные используются в выражения языка C, они автоматически преобразовываются и затем рассматриваются как целые числа.

В главе 6, “Принятие решений”, вы могли видеть, как выражение

```
c >= 'a' && c <= 'z'
```

использовалось для того, чтобы определить, является ли переменная символом, представляющим строчную букву. Как уже упоминалось, такое выражение можно использовать в тех случаях, когда для символов используется кодировка ASCII, в которой все строчные символы представлены последовательно, без наличия среди них других символов. В первой части представленного выражения производится проверка того, что значение переменной `c` превышает значение, соответствующее внутреннему коду для символа `'a'`. В кодировке ASCII для символа `'a'` установлено значение 97, для символа `'b'` установлено значение 98 и т.д.

Поэтому значение выражения `c>='a'` для всех значений переменной `c`, которые являются строчными буквами, будет равно `True`, поскольку все эти значения превышают или равны значению 97. Но поскольку в таблице кодов ASCII много других символов, превышающих значение 97 (например скобки), необходимо проверять и верхнюю границу диапазона, чтобы убедиться, что число попадает в заданный диапазон. По этой причине производится сравнение с символом `'z'`, цифровое представление которого в кодировке ASCII имеет значение 122.

Поскольку сравнение значения переменной `c` с символами `'a'` и `'z'` в предыдущем выражении на самом деле сравнивает значение переменной `c` с цифровым представлением символов `'a'` и `'z'`, то следующее выражение

```
c >= 97 && c <= 122
```

полностью эквивалентно предыдущему выражению и его можно использовать для выделения строчных символов. Но первое выражение будет предпочтительнее, поскольку при этом не требуется знания специфических числовых значений для символов `'a'` и `'z'` и выражение является более наглядным. Утверждение

```
printf ("%i\n", c);
```

можно использовать для вывода на экран значений, которые являются внутренним числовым представлением переменной *c*. Если используется кодировка ASCII, то, например, утверждение

```
printf ("%i\n", 'a');
```

отобразит число 97.

Постарайтесь предсказать, что будет отображено в результате выполнения следующих двух утверждений.

```
c = 'a' + 1;  
printf ("%c\n", c);
```

Поскольку цифровым представлением символа 'a' в кодировке ASCII является число 97, то в результате выполнения первого утверждения переменной *c* будет присвоено значение 98. Поскольку это значение в кодировке ASCII представляет символ 'b', то именно этот символ и будет отображен после выполнения второго утверждения.

Хотя добавление единицы к символьной константе кажется на очень практичным, но предшествующий пример открывает путь к важной концепции, используемой для преобразования символов от '0' до '9' в их соответствующие цифровые представления от 0 до 9. Вспомните, что символ '0' это не то же самое, что и целое число 0, символ '1' не является числом 1 и т.д. На самом деле символ '0' имеет числовое значение 48 в кодировке ASCII, которое можно отобразить с помощью вызова функции `printf`:

```
printf ("%i\n", '0');
```

Предположим, что символьная переменная *c* содержит один из символов от '0' до '9' и вы хотите преобразовать эти значение в соответствующее число от 0 до 9. Поскольку все символы цифр представляют последовательность чисел, вы можете легко преобразовать значение символьной переменной *c* в эквивалентное число путем простого вычитания символьной константы '0' из этого значения. Следовательно, если переменная *i* объявлена как целочисленная переменная, то утверждение

```
i = c - '0';
```

как раз и будет преобразовывать символ, содержащийся в переменной *c* в эквивалентное целое число. Предположим, *c* содержит символ '5', который в кодировке ASCII имеет цифровое значение 53. Цифровым значением символа '0' будет 48. Подставляя эти значения в предыдущее утверждение и вычитая 48 из 53, получим значение 5, которое и будет присвоено переменной *i*. На машине, использующей другую кодировку, не ASCII, вероятнее всего будет получен такой же результат, хотя внутреннее представление символов будет другое.

Такая техника может использоваться для преобразования символьных строк, состоящих из цифр, в их эквивалентное цифровое представление. Именно это и сделано в программе из листинга 10.11, в которой функция с именем `strToInt` служит для преобразования в целое число символьной строки, которая передается в нее в качестве аргумента. Функция прекращает сканирование строки в момент считывания нецифрового символа и возвращает результат в вызвавшую ее программу.

Предполагается, что целочисленная переменная имеет достаточно большой размер для того, чтобы сохранять преобразованные значения.

Листинг 10.11. Преобразование строки в ее числовой эквивалент

```
// Функция преобразования строки в число.
#include <stdio.h>
int strToInt (const char string[])
{
    int i, intValue, result = 0;

    for ( i = 0; string[i] >= '0' && string[i] <= '9'; ++i )
    {
        intValue = string[i] - '0';
        result = result * 10 + intValue;
    }
    return result;
}

int main (void)
{
    int strToInt (const char string[]);

    printf ("%i\n", strToInt("245"));
    printf ("%i\n", strToInt("100") + 25);
    printf ("%i\n", strToInt("13x5"));

    return 0;
}
```

Листинг 10.11. Вывод

```
245
125
13
```

Цикл `for` выполняется до тех пор, пока все символы для выражения `string[i]` являются цифровыми. К тому же при каждой итерации символ, содержащийся в `string[i]`, преобразовывается в эквивалентное числовое значение и затем добавляется к значению переменной `result`, умноженной на 10. Для того чтобы проанализировать, как это работает, рассмотрим пример вызова функции для символической строки "245". Сначала в цикле переменной `intValue` будет присвоено значение `string[0] - '0'`. Поскольку элемент `string[0]` содержит символ '2', то результирующее значение 2 будет присвоено переменной `intValue`. Поскольку сначала значение переменной `result` равно 0, то при первой итерации это значение, умноженное на 10, даст в результате 0, который и будет добавлен к переменной `intValue` и сохранен в переменной `result`. Таким образом, после первой итерации переменная `result` будет содержать значение 2.

При второй итерации переменная `intValue` примет значение 4, которое получится при вычитании символа '0' из символа '4'. Умножая `result` на 10, получим число 20, которое в сумме с переменной `intValue` даст значение 24, которое и будет сохранено в переменной `result`.

При третьей итерации переменная `intValue` примет значение '5' - '0', или 5, которое будет добавлено к значению переменной `result`, умноженной на 10 (240). Таким образом, значение переменной `result` после последнего цикла будет равно 245.

После считывания заключительного нулевого символа цикл `for` завершит работу и значение переменной `result`, число 245, будет возвращено в вызывающую программу. Функция `strToInt` может быть улучшена по двум направлениям. Во-первых, это обработка отрицательных чисел. Во-вторых, она должна контролировать принимаемую символьную строку и при наличии нецифровых символов возвращать значение 0. Например, если вызвать функцию следующим образом

```
strToInt ("xxx");
```

то должно быть получено значение 0. Эти улучшения вам предлагается выполнить в качестве упражнения.

На этом завершается обсуждение символьных строк. Как вы могли убедиться, язык программирования C предоставляет достаточно возможностей для удобной работы с символьными строками. Библиотеки содержат много функций для выполнения различных операций со строками. Например, имеется функция `strlen` для подсчета количества символов в символьной строке, функция `strcmp` — для сравнения двух строк, функция `strcat` для объединения двух строк, функция `strcpy` — для копирования одной строки в другую, функция `atoi` — для преобразования строки в целое число, а также функции `isupper`, `islower`, `isalpha` и `isdigit` — для проверки того, что все символы являются заглавными, строчными, буквами или цифрами соответственно. Хорошим упражнением будет переработка всех примеров из данной главы с использованием этих функций. В приложении Б, “Стандартная библиотека языка C”, перечислены многие функции из этой библиотеки.

Упражнения

1. Наберите и выполните 11 программ, представленных в этой главе. Сравните выходы, сделанные каждой программой, с выходами, представленными для соответствующих программ в книге.
2. Почему вы можете заменить утверждение `while` в функции `equalStrings` из листинга 10.4 следующим утверждением

```
while ( s1[i] == s2[i] && s1[i] != '\0' )
```

и получить аналогичный результат?

3. Функция `countWords` из программ 10.7 и 10.8 неправильно учитывает слова, которые содержат апострофы, как два отдельных слова. Модифицируйте эту функцию, чтобы она корректно обрабатывала такую ситуацию. Также расширьте функцию так, чтобы она учитывала последовательности положительных и отрицательных чисел, включая встроенные точки и запятые, как отдельные слова.
4. Напишите функцию с именем `substring`, которая будет извлекать часть символьной строки. Функция должна вызываться следующим образом

```
substring (source, start, count, result);
```

где параметр `source` является той строкой, из которой вы будете извлекать подстроку, параметр `start` задаст номер индекса в строке `source`, указывающий на первый символ подстроки, параметр `count` задает число извлекаемых символов из строки `source`, а параметр `result` является символьным массивом, который будет содержать извлеченную подстроку. Например, вызов функции следующим образом


```
substring ("character", 4, 3, result);
```

извлечет подстроку "act" (три символа начать с номера четыре) из строки "character" и поместит результат в массив result.

Не забудьте, что функция должна вставлять нулевой символ в конце подстроки в результирующем массиве. Также выполните проверку диапазона подстроки: если он выходит за пределы строки, то должна извлекаться подстрока только до конца строки. Поэтому при следующем вызове:

```
substring ("two words", 4, 20, result);
```

в результирующий массив должна быть помещена подстрока "words" несмотря на то, что требуется извлечь 29 символов.

5. Напишите функцию с именем `findString` для определения того, что одна строка существует в другой строке в качестве подстроки. Первый аргумент функции должен определять символьную строку, в которой должен производиться поиск, второй аргумент должен являться искомой подстрокой. Если функция находит подстроку, то она должна возвращать номер индекса в строке поиска, с которого начинается подстрока. Если искомая подстрока не найдена, то должно возвращаться значение -1. Поэтому, например, следующий вызов функции

```
index = findString ("a chatterbox", "hat");
```

должен произвести поиск подстроки "hat" в строке "a chatterbox". Поскольку такая подстрока находится в искомой строке, то функция должна вернуть значение 3, которое определяет начальную позицию искомой подстроки.

6. Напишите функцию с именем `removeString` для удаления заданного количества символов из символьной строки. Функция должна принимать три аргумента: исходная строка, номер начального индекса и количество удаляемых символов. Следовательно, если символьный массив `text` содержит строку "the wrong son", то вызов функции

```
removeString (text, 4, 6);
```

удалит из строки символы "wrong" (слово "wrong" и пробел за ним) из массива с именем `text`, после чего в массиве `text` должно остаться следующее: "the son".

7. Напишите функцию с именем `insertString` для вставки одной строки в другую. Параметры функции должны содержать исходную строку, вставляемую строку и номер индекса, с которого должна быть вставлена строка. Поэтому вызов функции

```
insertString (text, "per", 10);
```

вставит в символьный массив `text`, представленный в предыдущем упражнении, строку "per", которая будет быть вставлена в массив `text` с позиции 10. После выполнения функции, в массиве `text` должна находиться строка "the wrong person".

8. Используя функции `findString`, `removeString` и `insertString` из предыдущих упражнений, напишите функцию с именем `replaceString`, которая должна принимать три символьные строки в качестве аргументов следующим образом

```
replaceString (source, s1, s2);
```

и заменять подстроку `s1` внутри строки `source` на строку `s2`. Функция должна использовать функцию `findString` для поиска подстроки `s1` внутри строки `source`, и затем вызывать функцию `removeString` для удаления подстроки `s1` из строки `source`. Наконец функция `insertString` должна вставить строку `s2` в строку `source` с заданной позицией.

Таким образом, если произвести вызов

```
replaceString (text, "1", "one");
```

то первое вхождение подстроки "1" в строке `text`, если оно существует, будет заменено строкой "one". А если произвести следующий вызов функции

```
replaceString (text, "*", "");
```

то будет просто удален первый встретившийся символ звездочки, т.к. для его замены выбрана нулевая строка.

9. Вы можете расширить возможности функции `replaceString` из предыдущего упражнения, если заставите ее возвращать значение, свидетельствующее о том, что замена произведена успешно, т.е. что подстрока была найдена и заменена. Следовательно, если функция будет возвращать значение `true` при успешной замене и `false` при отсутствии замены, то вы можете написать цикл

```
do  
stillFound = replaceString (text, " ", "");  
while ( stillFound = true );
```

который, например, будет удалять все пробелы из символьной строки `text`. Дополните функцию `replaceStrings` и проверьте ее в различных ситуациях, чтобы убедиться в правильности ее работы.

10. Напишите функцию с именем `dictionarySort`, которая сортирует слова в словаре в алфавитном порядке, что используется в программах 10.9 и 10.10.
11. Расширьте функцию `strToInt` из листинга 10.11 таким образом, что если первым символом строки является знак минуса, то следующее за ним значение будет считаться отрицательным числом.
12. Напишите функцию с именем `strToFloat`, которая преобразовывает символьную строку в вещественное число. Функция должна учитывать знак минуса. Следовательно, если вызвать функцию следующим образом

```
strToFloat ("-867.6921");
```

то должно быть возвращено значение `-867.6921`.

13. Если символ `c` представляет строчную букву, то выражение:

```
c - 'a' + 'A'
```

возвратит эту букву как заглавную. При этом предполагается, что используется кодировка ASCII. Напишите функцию с именем `uppercase`, которая преобразовывает все строчные буквы в соответствующие заглавные буквы.

14. Напишите функцию с именем `intToStr`, которая преобразовывает целочисленное значение в символьную строку. Функция должна корректно обрабатывать отрицательные числа.

Указатели

В этой главе будут рассмотрены наиболее сложные конструкции языка программирования C: указатели. На самом деле мощь и гибкость языка C обеспечивается благодаря явному использованию указателей, в отличие от других языков программирования. Указатели позволяют эффективно использовать сложные структуры данных, изменять значения передаваемых в функцию в качестве аргументов переменных, работать с динамически размещаемой памятью (см. главу 17, "Дополнительные возможности") и более лаконично и просто использовать массивы.

Для понимания того, как работают указатели, сначала необходимо познакомиться с концепцией *косвенной адресации*. Вы используете такую концепцию каждый день. Например, предположим, что вы хотите купить новый чернильный картридж для своего принтера. В компании, где вы работаете, все покупки производятся через отдел снабжения. Поэтому вы звоните Джиму из этого отдела и заказываете ему новый картридж. Такая покупка картриджа на самом деле является косвенной покупкой, поскольку вы не обращаетесь непосредственно на склад, где эти картриджи находятся.

Аналогично действуют и указатели в языке C. Вы обращаетесь к указателю для того, чтобы получить значение отдельного элемента данных, на которое он указывает. Причина, по которой использование указателей выгоднее прямого обращения, та же, что и в приведенном примере покупки картриджа. Вы должны заниматься своей работой и вам не нужно знать все склады, где находятся картриджи. Этим занимается отдел снабжения и он это сделает лучше вас, а вы в случае необходимости обращаетесь только к ним. Поэтому использование указателей во многих случаях значительно облегчает работу.

Объявление указателей

Итак, рассмотрим, как работают указатели. Предположим, вы объявили переменную с именем `count` следующим образом.

```
int count = 10;
```

Вы также можете объявить еще одну переменную с именем `int_pointer`, которую можно использовать для косвенного доступа к переменной `count`, если объявить ее следующим образом.

```
int *int_pointer;
```

В языке программирования С звездочка используется для указания на то, что данная переменная имеет тип *указатель на int*. Это означает, что переменная `int_pointer` будет использоваться в программе для косвенной адресации к одному (или более) целочисленному значению.

В предыдущих программах вы уже познакомились с тем, как используется оператор `&` при вызове функции `scanf`. Это унарный оператор, который называется *оператором адресации* и который в языке программирования С используется для того, чтобы создать указатель. Следовательно, если `x` является переменной определенного типа, то выражение `&x` будет являться указателем на эту переменную. Такое выражение может быть присвоено любой переменной типа *указатель* (такую переменную можно называть просто указателем), если она объявлена как указатель на тот же самый тип, что и тип переменной `x`.

Поэтому, используя приведенные выше объявления переменных `count` и `int_pointer` можно написать следующее утверждение

```
int_pointer = &count;
```

для создания косвенной адресации между переменными `int_pointer` и `count`. В данном случае оператор адресации используется для присваивания переменной `int_pointer` адреса переменной `count`, а не значения, которое присвоено переменной `count`. То есть переменная `int_pointer` будет указывать на переменную `count`. Связь, которая образуется между переменными `int_pointer` и `count`, показана на рис. 11.1. Линия со стрелкой выражает ту идею, что переменная `int_pointer` прямо не содержит значение переменной `count`, а только указывает на эту переменную.

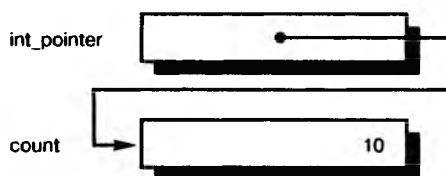


Рис. 11.1. Указатель на целое число

Сослаться на содержимое переменной `count` с помощью указателя `int_pointer` можно с помощью *оператора разыменования*, который изображается в виде звездочки `*`. Следовательно, если переменная `x` имеет тип `int`, то в утверждении

```
x = *int_pointer;
```

переменной `x` будет присвоено значение переменной, на которую указывает переменная `int_pointer`. Поскольку ранее мы сделали так, что переменная `int_pointer` стала указывать на переменную `count`, то в данном утверждении переменной `x` будет присвоено значение, которое содержится в переменной `count`. В нашем случае это значение 10.

Все предыдущие утверждения используются в программе из листинга 11.1, в котором демонстрируются два основных оператора для работы с указателями: оператор адресации `&` и оператор разыменования `*`.

Листинг 11.1. Демонстрация работы с указателями

```
// Программа для демонстрации работы с указателями.  
#include <stdio.h>  
int main (void)
```

```

{
    int count = 10, x;
    int *int_pointer;
    int_pointer = &count;
    x = *int_pointer;

    printf ("count = %i, x = %i\n", count, x);

    return 0;
}

```

Листинг 11.1. Вывод

```
count = 10, x = 10
```

Переменные `count` и `x` объявлены обычным способом как переменные целочисленного типа. В следующей строке переменная `int_pointer` объявлена как указатель на `int`. Обратите внимание, что две строки объявлений можно скомбинировать в одной строке.

```
int count = 10, x, *int_pointer;
```

Затем оператор адресации применяется в переменной `count`. При этом создается указатель на переменную, который затем присваивается переменной `int_pointer`.

Выполнение следующего утверждения программы

```
x = *int_pointer;
```

происходит следующим образом. Оператор разыменовывания заставляет компилятор языка C трактовать переменную `int_pointer` как содержащую указатель на другой элемент данных. Этот указатель затем используется для доступа к необходимым данным, тип которых должен совпадать с типом, на который указывает указатель. Поскольку компилятор уже знает из объявления, что переменная `int_pointer` является указателем на целочисленный тип данных, то он будет считывать целое значение. Но так как в предыдущих утверждениях программы было сделано так, чтобы указатель `int_pointer` стал указывать на целочисленную переменную `count`, то и будет произведен доступ к значению, содержащемуся в переменной `count`.

Вы должны понимать, что программа из листинга 11.1 только демонстрирует использование указателей и не имеет никакого практического применения. Более реальная программа будет представлена позже, когда вы научитесь увереннее обращаться с указателями.

В листинге 11.2 показаны некоторые интересные свойства указателей. Здесь используются указатели на символы.

Листинг 11.2. Базовые свойства указателей

```

// Примеры использования указателей
#include <stdio.h>
int main (void)
{
    char c = 'Q';
    char *char_pointer = &c;

    printf ("%c %c\n", c, *char_pointer);
}

```

```

c = '/';
printf ("%c %c\n", c, *char_pointer);
*char_pointer = '(';
printf ("%c %c\n", c, *char_pointer);

return 0;
}

```

Листинг 11.2. Вывод

```

Q Q
/ /
( (

```

В данной программе объявлена и инициализирована символом 'Q' символьная переменная `c`. В следующей строке программы объявлена переменная `char_pointer`, для которой выбран тип "указатель на `char`". Это означает, что любое значение, сохраненное в этой переменной, должно трактоваться как указатель (косвенная ссылка) на переменную символьного типа. Обратите внимание, что значение этой переменной можно присваивать обычным способом. Значение, которое в программе присвоено переменной `char_pointer`, будет являться указателем на переменную `c`, что достигается в помощью оператора адресации, используемого совместно с переменной `c`. (Учтите, что при инициализации возникнет ошибка компиляции, если переменная `c` будет объявлена после этого утверждения, поскольку переменная всегда должна быть объявлена до того, как она будет использоваться в утверждениях.)

Объявление переменной `char_pointer` и ее инициализация может быть записана двумя эквивалентными утверждениями, как это показано ниже.

```

char *char_pointer;
char_pointer = &c;

```

Но не подобными утверждениями

```

char *char_pointer;
*char_pointer = &c;

```

как это может показаться исходя из однострочного объявления и инициализации.

Всегда помните, что значение указателя в языке программирования C не будет определено, пока оно не будет задано явно.

При первом вызове процедуры `printf` будет просто отображено содержимое переменной `c` и содержимое переменной, на которую ссылается указатель `char_pointer`. Поскольку было задано, что переменная `char_pointer` должна указывать на переменную `c`, то будут выведены два одинаковых символа, что и подтверждается при выводе.

В следующей строке программы переменной `c` присваивается значение в виде символа. Поскольку переменная `char_pointer` все еще указывает на переменную `c`, то при вызове очередной процедуры `printf` будут также отображены одинаковые значения, которые в данном случае являются символами '/'. Это очень важно понять, пока значение переменной `char_pointer` не изменилось, значение выражения `*char_pointer` всегда будет указывать на переменную `c`. Поэтому при изменении значения переменной `c` будет изменено и значение выражения `*char_pointer`.

Исходя из предыдущих рассуждений, вы уже должны понять, что произойдет в следующем утверждении. Пока значение переменной `char_pointer` не изменится, выражение `*char_pointer` всегда указывает на переменную `c`. Поэтому в выражении

```
*char_pointer = '(';
```

значение '(' будет присвоено переменной `c`. Если же рассматривать вопрос более формально, то символ '(' будет присвоен переменной, на которую указывает переменная `char_pointer`. В нашей программе это будет переменная `c`, поскольку в начале программы указателю `char_pointer` был присвоен адрес переменной `c`.

Все рассмотренные концепции являются ключевыми для понимания работы указателей. Внимательно проанализируйте их еще раз и убедитесь, что вы все хорошо поняли.

Использование указателей в выражениях

В программе из листинга 11.3 заданы два целочисленных указателя, `p1` и `p2`. Обратите внимание, как значения, на которые ссылаются указатели, можно использовать в арифметических выражениях. Если переменная `p1` объявлена как "указатель на целое число", то как можно использовать комбинацию `*p1` в выражениях?

Листинг 11.3. Использование указателей в выражениях

```
// Еще об указателях.
#include <stdio.h>
int main (void)
{
    int i1, i2;
    int *p1, *p2;

    i1 = 5;
    p1 = &i1;
    i2 = *p1 / 2 + 10;
    p2 = p1;
    printf ("i1 = %i, i2 = %i, *p1 = %i, *p2 = %i\n", i1, i2, *p1, *p2);

    return 0;
}
```

Листинг 11.3 Вывод

```
i1 = 5, i2 = 12, *p1 = 5, *p2 = 5
```

После объявления целочисленных переменных `i1` и `i2` и указателей на целые числа `p1` и `p2`, в программе присваивается значение 5 переменной `i1` и сохраняется ссылка на `i1` в указателе `p1`. Затем рассчитывается значение переменной `i2` с помощью следующего утверждения.

```
i2 = *p1 / 2 + 10;
```

Как уже говорилось при обсуждении программы из листинга 11.2, если указатель `rx` указывает на переменную `x` и объявленный тип, на который указывает указатель `rx`, совпадает с типом переменной `x`, то тогда использование комбинации `*rx` в выражении будет аналогично использованию одной переменной `x`.

Поскольку в листинге 11.3 переменная `p1` объявлена как указатель на целое число, то предыдущее выражение будет вычисляться по правилам целочисленной арифметики. А поскольку значением выражения `*p1` является число 5 (`p1` указывает на `i1`), то

окончательным результатом расчета выражения будет число 12, которое и будет присвоено переменной `i2`. (Здесь необходимо учитывать, что оператор разыменовывания `**` имеет высший приоритет, чем операция арифметического деления. На самом деле, этот оператор, как и оператор адресации `&`, имеет приоритет над всеми бинарными операторами в языке программирования C).

В следующем утверждении значение указателя `p1` присваивается указателю `p2`. Такое присваивание вполне допустимо и при этом значение указателя `p2` будет указывать на ту же переменную, что и указатель `p1`. Поскольку `p1` указывает на переменную `i1`, то после присваивания указатель `p2` также будет указывать на переменную `i1` (вы можете иметь сколь угодно много указателей на одну и ту же переменную).

Вывод, сделанный с помощью процедуры `printf` подтверждает, что значения для выражений `i1`, `*p1` и `*p2` совпадают (число 5) и что значением переменной `i2` будет число 12.

Указатели на структуры

Как вы уже могли убедиться, указатели можно объявлять таким образом, чтобы они указывали на базовые типы данных, такие как `int` или `char`. Но указатели можно объявлять и так, что они будут ссылаться на структуры. В главе 9, “Структуры”, структуры объявлялись следующим образом

```
struct date
{
    int month;
    int day;
    int year;
};
```

После того как будет объявлена переменная типа `struct date`, что можно сделать следующим образом

```
struct date todaysDate;
```

вы можете объявить переменную, которая будет указывать на тип `struct date`.

```
struct date *datePtr;
```

После такого объявления указателя `datePtr` его можно использовать обычным способом. Например, можно сделать так, что он будет указывать на структуру `todaysDate`, для чего необходимо написать следующее утверждение.

```
datePtr = &todaysDate;
```

После такого присваивания можно косвенно обращаться к любому члену структуры `date` с помощью указателя `datePtr` следующим образом.

```
(*datePtr).day = 21;
```

Такое утверждение можно использовать для того, чтобы присвоить члену структуры `day`, на которую ссылается указатель `datePtr`, значение 21. В данном случае круглые скобки необходимы, поскольку оператор выделения члена структуры `“.”` имеет высший приоритет, чем оператор разыменовывания `**`.

Для проверки того, что определенное значение присвоено члену `month` структуры `date`, на которую ссылается указатель `datePtr`, необходимо написать следующее.

```
if ( (*datePtr).month == 12 )
```

```
...
```

Поскольку структуры довольно часто используются в языке программирования C, то для работы с ними выделен специальный оператор. Оператор “->”, который представляет комбинацию тире и символа “больше”, необходимо рассматривать как ссылку на структуру, и с его помощью выражение.

```
(*x).y
```

может быть записано в следующем виде.

```
x->y
```

Поэтому предыдущее утверждение if можно более удобно и понятно записать в следующем виде.

```
if ( datePtr->month == 12 )
```

```
...
```

Программа из листинга 11.4 представляет собой программу из листинга 9.1, переписанную с использованием указателей на структуры.

Листинг 11.4. Использование указателей на структуры

```
// Программа демонстрирует использование указателей на структуры.
#include <stdio.h>
int main (void)
{
    struct date
    {
        int month;
        int day;
        int year;
    };
    struct date today, *datePtr;

    datePtr = &today;
    datePtr->month = 9;
    datePtr->day = 25;
    datePtr->year = 2004;
    printf ("Today's date is %i/%i/%.2i.\n",
        datePtr->month, datePtr->day, datePtr->year % 100);

    return 0;
}
```

Листинг 11.4. Вывод

```
Today's date is 9/25/04.
```

На рис. 11.2 показано, как будут выглядеть переменные today и datePtr после всех присваиваний, сделанных в предыдущей программе.

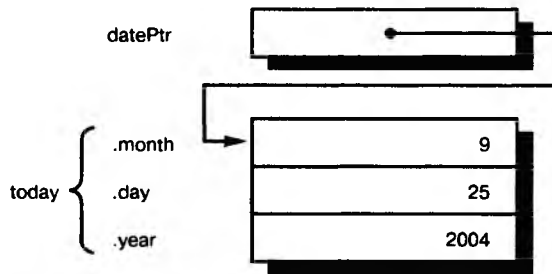


Рис. 11.2. Указатель на структуру

Считаю не лишним напомнить еще раз, что все предыдущие примеры являются только демонстрацией использования указателей на структуры, и вы должны хорошо понять, как они работают, прежде чем использовать указатели в реальных задачах.

Указатели в структурах

Вполне естественно, что и сами указатели могут быть членами структуры. В определении структуры:

```
struct intPtrs
{
    int *p1;
    int *p2;
};
```

описана структура с именем `intPtrs`, которая содержит два указателя на целые числа: первый имеет имя `p1`, а второй назван `p2`. После этого описания можно объявить переменную типа `struct intPtrs` следующим образом.

```
struct intPtrs pointers;
```

После объявления переменная `pointers` может использоваться как обычный указатель, который будет ссылаться на структурную переменную, содержащую два указателя в качестве членов. В программе из листинга 11.5 показано, как можно обрабатывать структуру `intPtrs` на языке C.

Листинг 11.5. Использование структур, содержащих указатели

```
// Функция для работы со структурами, содержащими указатели.
#include <stdio.h>
int main (void)
{
    struct intPtrs
    {
        int *p1;
        int *p2;
    };
    struct intPtrs pointers;

    int i1 = 100, i2;
    pointers.p1 = &i1;
    pointers.p2 = &i2;
```

```

*pointers.p2 = -97;
printf ("i1 = %i, *pointers.p1 = %i\n", i1, *pointers.p1);
printf ("i2 = %i, *pointers.p2 = %i\n", i2, *pointers.p2);

return 0;
}

```

Листинг 11.5 Вывод

```

i1 = 100, *pointers.p1 = 100
i2 = -97, *pointers.p2 = -97

```

После объявления переменных записывается утверждение присваивания

```
pointers.p1 = &i1;
```

в котором члену p1 структуры pointers присваивается адрес переменной i1, а в следующем утверждении

```
pointers.p2 = &i2;
```

члену структуры p2 присваивается адрес переменной i2. Затем значение -97 присваивается переменной, на которую указывает член pointers.p2. Поскольку мы только что сделали так, чтобы он указывает на i2, то значение -97 сохраняется в переменной i2. Никаких скобок в данном случае не требуется, т.к. оператор выделения члена структуры "." имеет более высокий приоритет, чем оператор разыменовывания.

Поэтому указатель будет корректно ссылаться на структуру до того, как будет применен оператор разыменовывания. Разумеется, скобки можно использовать, но только для того, чтобы застраховать себя от неприятностей, если вы не помните точно приоритеты операторов. Две процедуры printf вызываются для того, чтобы проверить правильность наших рассуждений и проконтролировать результаты присваиваний.

На рис. 11.3 показаны взаимоотношения между переменными i1, i2 и pointers после того, как были выполнены утверждения присваивания в программе из листинга 11.5. Как вы можете видеть из рисунка, член структуры p1 указывает на переменную i1, которая содержит значение 100, а член p2 указывает на переменную i2, которая содержит значение -97.

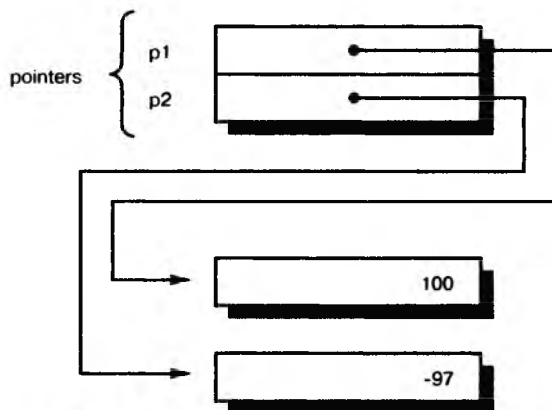


Рис. 11.3. Структура, содержащая указатели

Связанные списки

Указатели на структуры и структуры, содержащих указатели, являются довольно мощным инструментом в языке программирования С и они используются в таких изощренных конструкциях, как *связанные списки*, *двойные связанные списки* и *деревья*.

Предположим, вы описали структуру следующим образом:

```
struct entry
{
    int value;
    struct entry *next;
};
```

Здесь описана структура с именем `entry`, которая имеет два члена. Первый член структуры является простым целым числом с именем `value`, а второй член имеет имя `next` и является указателем на структуру `entry`. Обратите внимание на то, что содержащийся внутри структуры `entry` указатель ссылается на структуру `entry`.

В языке программирования С это вполне допустимая конструкция. Сейчас предположим, что вы объявили две переменные типа `struct entry` следующим образом

```
struct entry n1, n2;
```

Затем вы установили указатель структуры `n1` так, что он стал указывать на структуру `n2` с помощью следующего утверждения.

```
n1.next = &n2;
```

С помощью такого утверждения вы создадите связь между структурами `n1` и `n2`, как показано на рис. 11.4.

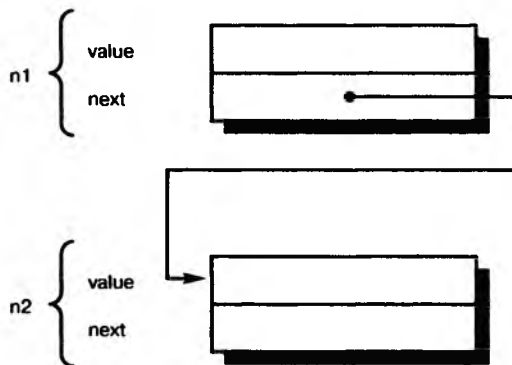


Рис. 11.4. Связанные структуры

Присваивая адрес переменной `n3`, которая также объявлена как тип `struct entry`, можно создать дополнительную связь с помощью следующего утверждения

```
n2.next = &n3;
```

В результате будет получена цепь связанных элементов, которая обычно называется *связанным списком*. Его взаимосвязи показаны на рис. 11.5. В программе из листинга 11.6 показан пример создания связанного списка.

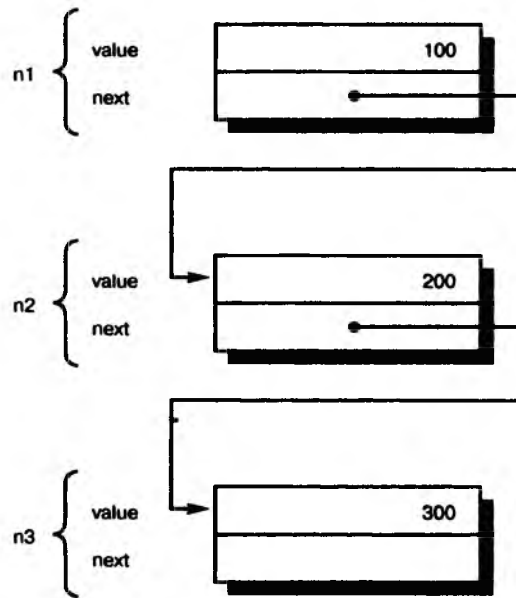


Рис. 11.5. Связанный список

Листинг 11.6. Использование связанных списков

```
// Function to use linked Lists
#include <stdio.h>
int main (void)
{
    struct entry
    {
        int value;
        struct entry *next;
    };
    struct entry n1, n2, n3;
    int i;

    n1.value = 100;
    n2.value = 200;
    n3.value = 300;
    n1.next = &n2;
    n2.next = &n3;
    i = n1.next->value;
    printf ("%i ", i);
    printf ("%i\n", n2.next->value);

    return 0;
}
```

Листинг 11.6 Вывод

200 300

Структуры `n1`, `n2` и `n3` объявлены как тип `struct entry`, который состоит из целочисленного члена с именем `value` и указателя на структуру `entry` с именем `next`. Затем в программе присваиваются значения 100, 200 и 300 целочисленным членам структур `n1`, `n2` и `n3` соответственно.

Следующие два утверждения программы

```
n1.next = &n2;
n2.next = &n3;
```

создают связанный список с помощью члена `next` структуры `n1`, указывающего на структуру `n2`, и члена `next` структуры `n2`, указывающего на структуру `n3`.

Выполнение утверждения

```
i = n1.next->value;
```

приводит к следующему: значение члена `value` структуры `entry`, на которую указывает выражение `n1.next`, будет присвоено целочисленной переменной `i`. Поскольку выражение `n1.next` указывает на структуру `n2`, то будет использовано значение члена `value` структуры `n2`. Поэтому результатом вычисления этого выражения будет присваивание значения 200 переменной `i`, что и подтверждается после вывода, сделанного с помощью процедуры `printf`.

Вы также можете убедиться, что выражение `n1.next->value` выполняется в соответствии с указанным синтаксисом, причем это не одно и то же, что `n1.next.value`, поскольку поле `n1.next` содержит указатель на структуру, а не саму структуру. Это различие очень важно и часто приводит к ошибкам в программе из-за недостаточного понимания этой особенности.

Оператор выделения члена структуры `"."` и указатель на структуру `"->"` в языке программирования C имеют одинаковый приоритет. Поэтому в таком выражении, как предыдущее, где используются оба эти оператора, вычисление производится последовательно слева направо. То есть вычисление производится следующим образом.

```
i = (n1.next)->value;
```

Вторичный вызов процедуры `printf` в листинге 11.6 приводит к отображению значения члена `value`, на который указывает выражение `n2.next`. Поскольку это выражение указывает на структуру `n3`, то отображается содержимое члена `n3.value`.

Как уже упоминалось, концепция связанных списков является очень мощной в программировании. Связанные списки значительно облегчают выполнение таких операций, как вставка и удаление элементов из большого списка и их сортировку.

Например, если взять структуры `n1`, `n2` и `n3`, которые мы только что использовали, то можно очень легко удалить из списка структуру `n2`, просто установив для поля `next` структуры `n1` значение, которое будет указывать на структуру `n3`. Это можно сделать с помощью следующего утверждения

```
n1.next = n2.next;
```

Здесь производится копирование указателя `n2.next`, который указывает на структуру `n3`, в указатель `n1.next`, после чего `n1.next` будет указывать на структуру `n3`.

Поскольку теперь больше ничего не указывает на структуру `n2`, то получен эффект удаления ее из списка. На рис. 11.6 показана ситуация после выполнения вышеприведенного утверждения. Разумеется, можно непосредственно установить указатель структуры `n1` так, чтобы он указывал на структуру `n3`, что и сделано с следующим утверждением.

```
n1.next = &n3;
```

Но лучше не использовать такие утверждения, поскольку вы должны заранее знать, что `n2` указывает на `n3`.

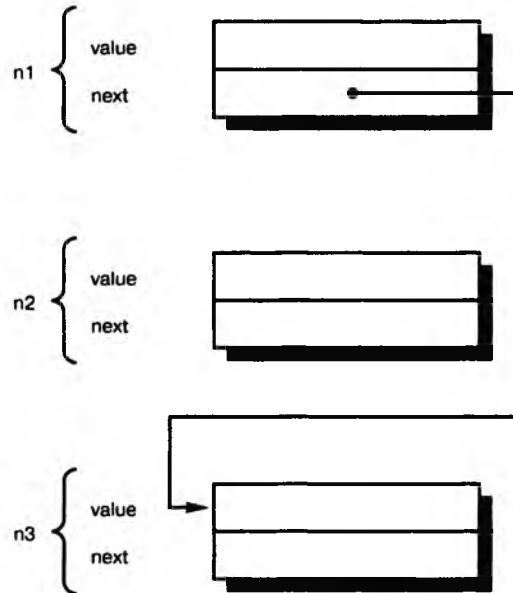


Рис. 11.6. Удаление элемента из связанного списка

Вставить элемент в список тоже довольно просто. Если вы хотите вставить элемент типа `struct entry` с именем `n2_3` после элемента `n2` в списке, то вы должны сделать так, чтобы указатель `n2_3.next` указывал туда, куда указывает указатель `n2.next`, а затем установить указатель `n2.next` так, чтобы он указывал на `n2_3`. Поэтому последовательность утверждений

```
n2_3.next = n2.next;
n2.next = &n2_3;
```

вставит структуру `n2_3` непосредственно после `n2`. Обратите внимание, что последовательность предыдущих операторов крайне важна, поскольку выполнение второго утверждения перед первым приведет к тому, что указателю `n2.next` будет присвоено значение `n2_3.next` до того, как оно будет изменено. Вставка элемента изображена на рис. 11.7. Заметьте, что структура `n2_3` изображена не между структурами `n1` и `n3`. Это подчеркивает тот факт, что в памяти структура `n2_3` может располагаться где угодно, а не только после структуры `n1` и перед структурой `n3`. Это является основным поводом для использования связанных списков при хранении информации. Элементы списка не должны храниться в последовательном порядке, как это происходит при хранении массивов.

До того, как перейдем к разработке функции для работы со связанными списками, обсудим еще некоторые проблемы. Обычно для обращения к связанному списку используется указатель на список. Чаще всего этот указатель ссылается на первый элемент списка. Поэтому для нашего трехэлементного списка, который состоит из структур `n1`, `n2` и `n3`, можно задать переменную с именем `list_pointer` и сделать так, чтобы она указывала на начало списка с помощью следующего утверждения


```
struct entry *list_pointer = &n1;
```

где предполагается, что структура n1 уже объявлена. Указатель на список можно использовать для последовательного просмотра всех элементов списка, чем мы очень скоро будем пользоваться.

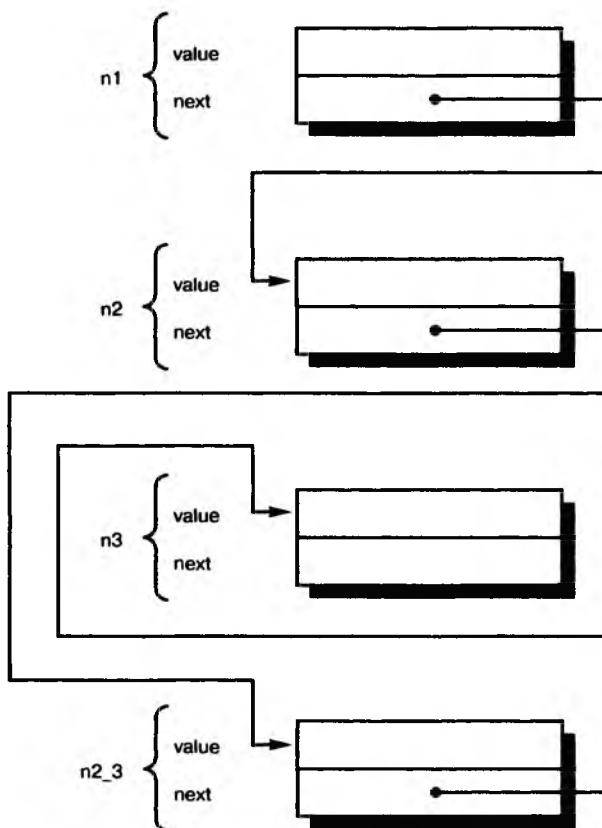


Рис. 11.7. Вставка элемента в связанный список

Вторая проблема, которую необходимо обсудить, касается обнаружения конца связанного списка. Это необходимо для того, чтобы, например, процедура, производящая просмотр элементов списка, могла определить конечный элемент списка. По соглашению, для конечного элемента используется константное значение 0, которое называют *нулевым указателем* (null). Нулевой указатель можно использовать для выделения конца списка, сохраняя это значение в поле указателя для последнего элемента списка.

В нашем случае с тремя элементами, мы можем отметить конец связанного списка с помощью нулевого указателя в поле n3.next следующим образом:

```
n3.next = (struct entry *) 0;
```

В главе 13, “Препроцессоры” вы познакомитесь с тем, как сделать утверждения присваивания более читабельным. В данном утверждении выполняется приведение типов для того, чтобы использовать константу 0 как необходимый тип (“указатель на тип struct entry”). Хотя это и не обязательно, но при этом получается более понятное утверждение.

На рис. 11.8 изображен связанный список из листинга 11.7, в котором указатель на тип `struct entry` с именем `list_pointer` указывает на начальный элемент списка, а в поле `n3.next` записывается нулевой указатель.

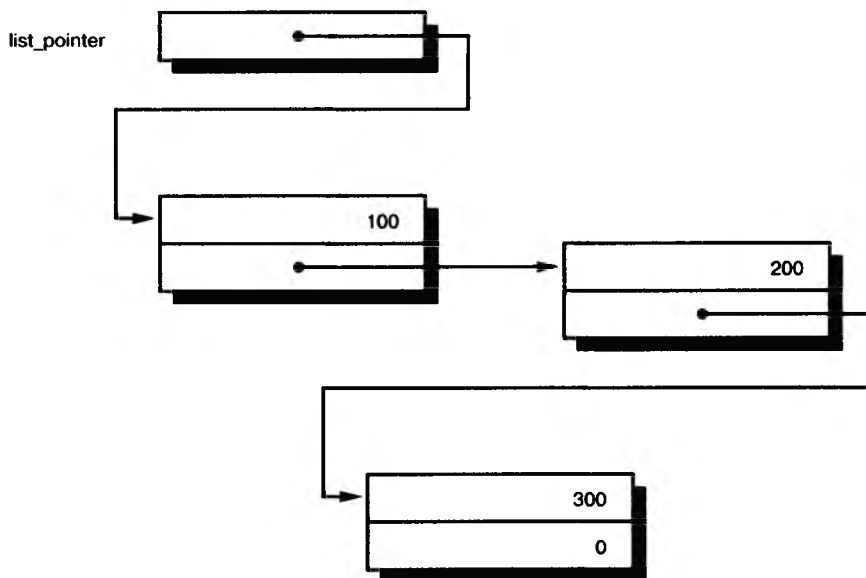


Рис. 11.8. Связанный список с указателем на список и конечным нулевым указателем

Программа из листинга 11.7 использует все только что обсуждаемые темы. Для последовательного просмотра элементов списка и отображения значения члена `value` каждого элемента списка в программе используется цикл `while`

Листинг 11.7. Просмотр связанного списка

```
// Программа для просмотра связанного списка.
#include <stdio.h>
int main (void)
{
    struct entry
    {
        int value;
        struct entry *next;
    };
    struct entry n1, n2, n3;
    struct entry *list_pointer = &n1;

    n1.value = 100;
    n1.next = &n2;
    n2.value = 200;
    n2.next = &n3;
    n3.value = 300;
    n3.next = (struct entry *) 0; // Закончить список нулевым
    указателем.
    while ( list_pointer != (struct entry *) 0 ) {
        printf ("%i\n", list_pointer->value);
        list_pointer = list_pointer->next;
    }
}
```

```
}  
return 0;  
}
```

Листинг 11.7. Вывод

```
100  
200  
300
```

В программе заданы переменные `n1`, `n2` и `n3` и указатель `list_pointer`, который изначально устанавливается так, чтобы указывать на переменную `n1` — первый элемент списка. В следующих утверждениях программы связываются три элемента списка, а для элемента списка `n3` устанавливается нулевой указатель.

Затем используется цикл `while` для последовательного просмотра элементов списка. Цикл выполняется до тех пор, пока значение переменной `list_pointer` не равно нулевому указателю. Процедура `printf` вызывается внутри списка для того, чтобы отобразить значение поля `value` структуры, на которую в данный момент ссылается указатель `list_pointer`.

Следующее после вызова процедуры `printf` утверждение

```
list_pointer = list_pointer->next;
```

присваивает указателю `list_pointer` значение следующего элемента списка. При этом во время первой итерации в этом утверждении используется указатель, содержащийся в `n1.next` (т.к. переменная `list_pointer` инициализируется значением, указывающим на `n1`), который и присваивается переменной `list_pointer`. Поскольку это значение не равно нулю (оно указывает на элемент списка `n2`), цикл `while` продолжает выполняться.

При второй итерации производится отображения значения `n2.value`, которое равно 200. Значение члена `next` структуры `n2` затем копируется в переменную `list_pointer`, и поскольку это значение является ссылкой на элемент `n3`, то указатель `list_pointer` будет ссылаться на `n3` после второй итерации цикла.

Когда цикл `while` выполнится в третий раз, процедура `printf` отобразит значение 300, содержащееся в поле `n3.value`. После этого значение выражения `list_pointer->next` (которое равно `n3.next`) копируется в переменную `list_pointer`, и поскольку это будет нулевой указатель, то после трех итераций происходит выход из цикла `while`.

Проверьте все операции цикла `while` с помощью карандаша и бумаги, и если вам не все абсолютно понятно, то последовательно записывайте значения всех переменных. Понимание всех операций этого цикла является ключом к пониманию работы указателей в языке программирования C. При этом отметьте, что этот же самый цикл `while` может использоваться для просмотра связанного списка любого размера, т.е. конец списка отмечен нулевым указателем.

Когда вы работаете с реальным связанным списком в программе, то обычно вы не имеете явно заданных членов (как это делалось в предыдущих примерах). В предыдущих примерах был только продемонстрирован механизм работы связанных списков. На практике чаще всего вы будете запрашивать систему на выделение необходимой памяти для каждого нового элемента, который вы собираетесь вставить в список. Это делается с помощью механизма, известного как *динамическое выделение памяти*, и об этом подробно будет рассказано в главе 17.

Ключевое слово `const` и указатели

Вы уже знаете, что переменные или массивы могут объявляться с ключевым словом `const` для того, чтобы предупредить компилятор, а заодно и пользователя о том, что содержимое этих переменных или массивов не может изменяться в процессе выполнения программы. С указателями может возникнуть неоднозначная ситуация: что именно не должно изменяться. Или не должен изменяться сам указатель, или переменная, на которую он указывает. Подумайте над этим. Предположим, что сделано следующее объявление.

```
char c = 'X';  
char *charPtr = &c;
```

После этих утверждений указатель `charPtr` будет указывать на переменную `c`. Если указатель должен всегда указывать на переменную `c`, то он должен быть объявлен с ключевым словом `const` следующим образом:

```
char * const charPtr = &c;
```

Можно сказать, что "объявлен константный указатель на символ". Поэтому утверждение, подобное следующему

```
charPtr = &d; // Не разрешено.
```

приведет тому, что компилятор GNU выдаст сообщение:

```
foo.c:10: warning: assignment of read-only variable 'charPtr'  
(foo.c:10: предупреждение: присваивание делается для переменной  
'charPtr', которая предназначена только для чтения)
```

Другие компиляторы отобразят похожее сообщение, а некоторые из них могут не отреагировать совсем.

Если же вместо того, чтобы делать константным указатель, необходимо сделать неизменным значение, на которое он указывает, то объявление должно быть выполнено следующим образом:

```
const char *charPtr = &c;
```

Можно сказать, что "указатель `charPtr` указывает на константное значение". Но это не означает, что значение переменной `c`, на которую ссылается указатель `charPtr`, не может быть изменено. Это означает только то, что оно не может быть изменено с помощью утверждений, подобных следующему.

```
*charPtr = 'Y'; // Не разрешено.
```

В этом случае компилятор GNU выдаст следующее сообщение

```
foo.c:11: warning: assignment of read-only location  
(foo.c:11: warning: присваивание для ссылки, предназначенной только  
для чтения)
```

Если же необходимо, чтобы не изменялись ни указатель, ни переменная, на которую он указывает, то можно написать следующее объявление

```
const char * const *charPtr = &c;
```

Первое ключевое слово `const` говорит о том, что значение, на которое ссылается указатель, не должно изменяться. Второе слово `const` используется для того, чтобы сделать неизменным сам указатель. Хотя это и выглядит несколько запутанно, но зато выполняются необходимые требования. Поэтому в примерах программ не всегда используются ключевые слова `const` там, где из можно поставить. Пока вы не научитесь свободно понимать выражения с такими ключевыми словами, вы будете испытывать затруднения при чтении листингов.

Указатели и функции

Указатели и функции корректно работают совместно. Поэтому можно передавать указатели в функцию в качестве аргументов и принимать указатели, возвращаемые функцией в качестве результирующего значения.

Первый случай, т.е. передача указателей в качестве аргументов, не вызывает затруднений и хорошо вписывается в логику общих рассуждений об аргументах. Указатели просто перечисляются в списке аргументов для функции обычным способом. Поэтому для того, чтобы передать указатель `list_pointer` из предыдущей программы в функцию с именем `print_list`, вы просто пишете следующее.

```
print_list (list_pointer);
```

При этом, в описании функции `print_list` в качестве формального параметра должен быть задан указатель на соответствующий тип.

```
void print_list (struct entry *pointer)
{
    ...
}
```

Формальный параметр `pointer` может использоваться точно так же, как обычный указатель. В данном случае необходимо хорошо представлять последовательность передачи значений в параметр при вызове функции. Значение указателя просто копируется в формальный параметр, поэтому все изменения, которые будут производиться с формальным параметром в функции, не будут отражаться на указателе, который передавался в качестве аргумента. При этом необходимо понимать, что хотя нельзя изменить исходный указатель, можно изменить значение, на которые он указывает! Изучив программу из листинга 11.8, вы лучше поймете это.

Листинг 11.8. Использование указателей и функций

```
// Программа иллюстрирует использование указателей и функций.
#include <stdio.h>
void test (int *int_pointer)
{
    *int_pointer = 100;
}
int main (void)
{
    void test (int *int_pointer);
    int i = 50, *p = &i;

    printf ("Before the call to test i = %i\n", i);
    test (p);
```

```
printf ("After the call to test i = %i\n", i);  
return 0;  
}
```

Листинг 11.8. Вывод

```
Before the call to test i = 50  
After the call to test i = 100
```

В функции `test` в качестве параметра используется указатель на целое число. В теле функции выполняется одно утверждение, в котором присваивается значение 100 переменной, на которую ссылается указатель `int_pointer`.

В функции `main` объявляется целочисленная переменная `i`, которая инициализируется значением 50. Также объявляется указатель `p` на целочисленную переменную, которому присваивается адрес переменной `i`. Как можно видеть из второй строки, выведенной программой, функция `test` действительно изменяет значение переменной `i` с 50 на 100. Далее рассмотрим программу из листинга 11.9.

Листинг 11.9. Использование указателей для изменения значений

```
// Еще об указателях и функциях.  
#include <stdio.h>  
void exchange (int * const pint1, int * const pint2)  
{  
    int temp;  
  
    temp = *pint1;  
    *pint1 = *pint2;  
    *pint2 = temp;  
}  
  
int main (void)  
{  
    void exchange (int * const pint1, int * const pint2);  
    int i1 = -5, i2 = 66, *p1 = &i1, *p2 = &i2;  
    printf ("i1 = %i, i2 = %i\n", i1, i2);  
    exchange (p1, p2);  
    printf ("i1 = %i, i2 = %i\n", i1, i2);  
    exchange (&i1, &i2);  
    printf ("i1 = %i, i2 = %i\n", i1, i2);  
  
    return 0;  
}
```

Листинг 11.9. Вывод

```
i1 = -5, i2 = 66  
i1 = 66, i2 = -5  
i1 = -5, i2 = 66
```

Функция `exchange` должна переставлять значения двух переменных, на которые указывают ее аргументы. Из заголовка функции `exchange`

```
void exchange (int * const pint1, int * const pint2)
```

видно, что функция в качестве аргументов принимает два указателя на целые числа и эти указатели не могут быть изменены в функции (для чего используется ключевое слово `const`).

Локальная целочисленная переменная `temp` используется для сохранения значения одного из чисел во время обмена значений. Ей присваивается значение целочисленной переменной, на которую ссылается указатель `pint1`. Значение переменной, на которую ссылается указатель `pint2`, затем присваивается переменной, на которую ссылается указатель `pint1`, а значение переменной `temp` помещается в целочисленную переменную, на которую ссылается указатель `pint2`. На этом обмен значениями завершается.

В подпрограмме `main` для целочисленных переменных `i1` и `i2` задаются значения -5 и 66 соответственно. Двум указателям на целые числа `p1` и `p2` присваиваются адреса переменных `i1` и `i2` соответственно. Затем в программе отображаются значения переменных `i1` и `i2` и вызывается функция `exchange`, в которую в качестве аргументов передаются указатели на эти переменные. Функция `exchange` переставляет значения этих переменных, для чего используются указатели `p1` и `p2`. Поскольку `p1` указывает на `i1`, а `p2` на `i2`, то происходит перестановка значений для этих переменных. В этом можно убедиться при втором выводе переменных с помощью функции `printf`.

Второй вызов функции `exchange` более интересен. Поскольку аргументы, передаваемые в функцию, должны являться указателями на переменные `i1` и `i2`, т.е. адресами этих переменных, то используя оператор адресации, можно также получить адреса переменных. Поскольку в результате вычисления выражения `&i1` получается адрес этой переменной, или указатель на переменную `i1`, то это и будет соответствовать необходимому типу аргумента (указатель на целое число). Аналогичная ситуация наблюдается и со вторым аргументом, поэтому происходит корректное выполнение функции `exchange` и изменение значений этих переменных, т.е. они получают свои первоначальные значения.

Вы должны хорошо понимать, что без использования указателей невозможно создать функцию, которая переставляла бы значения двух целочисленных переменных, поскольку функция может возвращать только одно значение, а не два. Изучите детально программу из листинга 11.9. В этом небольшом примере используются ключевые концепции, понимание которых позволит вам легко оперировать указателями, программируя на языке C.

В программе из листинга 11.10 показан возврат функцией указателя. Здесь описана функция с именем `findEntry`, которая должна найти в связанном списке заданное значение. Когда нужное значение найдено, программа возвращает указатель на элемент списка, содержащий это значение. Если же заданное значение не найдено, функция возвращает нулевой указатель.

Листинг 11.10. Возврат указателя из функции

```
#include <stdio.h>
struct entry
{
    int value;
    struct entry *next;
};
struct entry *findEntry (struct entry *listPtr, int match)
{
    while ( listPtr != (struct entry *) 0 )
        if ( listPtr->value == match )
```

```

    return (listPtr);
else
    listPtr = listPtr->next;

return (struct entry *) 0;
}

int main (void)
{
    struct entry *findEntry (struct entry *listPtr, int match);
    struct entry n1, n2, n3;
    struct entry *listPtr, *listStart = &n1;
    int search;

    n1.value = 100;
    n1.next = &n2;
    n2.value = 200;
    n2.next = &n3;
    n3.value = 300;
    n3.next = 0;
    printf ("Enter value to locate: ");
    scanf ("%i", &search);
    listPtr = findEntry (listStart, search);
    if ( listPtr != (struct entry *) 0 )
        printf ("Found %i.\n", listPtr->value);
    else
        printf ("Not found.\n");

return 0;
}

```

Листинг 11.10. Вывод

```

Enter value to locate: 200
Found 200.

```

Листинг 11.10. Вывод (Повторение)

```

Enter value to locate: 400
Not found.

```

Листинг 11.10. Вывод (Второе повторение)

```

Enter value to locate: 300
Found 300.

```

Из заголовка функции

```
struct entry *findEntry (struct entry *listPtr, int match)
```

видно, что задается функция findEntry, которая возвращает указатель на структуру entry и в которой используются два параметра: указатель на структуру entry и целочисленная переменная. Тело функции начинается с цикла while, в котором последовательно просматриваются все элементы списка. Цикл выполняется до тех пор, пока не

будет найдено совпадение одного из элементов списка со значением параметра `match` (в этом случае выполнение цикла прекращается и возвращается значение `listPtr`), или не будет достигнут конец списка (в этом случае просматривается весь список и возвращается нулевой указатель).

После установки значений списка в программе, производится запрос к пользователю на ввод значения, затем вызывается функция `findEntry`, в которую передаются в качестве аргументов указатель на начальный элемент списка и значение переменной `search`, введенное пользователем.

Указатель, который возвращается функцией `findEntry`, присваивается переменной `listPtr` типа `struct entry`. Если значение указателя переменной `listPtr` не является нулем, то отображается член структуры `value` этой переменной. Оно должно быть таким же, как значение, введенное пользователем. Если указатель структуры `listPtr` равен нулю, то отображается сообщение "Not found" (Не найдено).

Из вывода, сделанного программой, можно убедиться в том, что числа 200 и 300 находятся в списке, а число 400 не найдено, поскольку его нет в списке.

В данной программе указатель, который возвращается функцией `findEntry` не выполняет никаких полезных функций. Однако на практике его можно использовать для доступа к другим элементам списка. Например, вы можете создать связанный список для словаря, используя знания, полученные в главе 10 "Символьные строки". Затем вы можете использовать функцию `findEntry` (или переименовать ее в `lookup`, как это сделано в этой главе) для поиска в словаре заданных слов. Указатель, который возвращается функцией `lookup`, может использоваться для доступа к члену структуры `definition` найденного элемента.

Организация словаря в виде связанного списка имеет несколько преимуществ. Например, очень легко вставлять новые слова: после определения места, куда надо вставить слово, необходимо просто изменить некоторые указатели, что и было показано несколько ранее в этой главе. Удаление элемента списка также не представляет трудностей. Наконец, из главы 17 вы узнаете, как эти и другие возможности поддерживаются средой разработки, что позволяет, например, динамически изменять размеры словаря.

Однако организация словаря в виде связанного списка имеет и один существенный недостаток. Он не позволяет использовать двоичный алгоритм для поиска элементов в таком списке. Этот алгоритм работает только с массивами, которые можно непосредственно индексировать. К сожалению, самым быстрым поиском для связанного списка может быть только последовательный перебор элементов списка, пока не будет обнаружено заданное значение, т.к. доступ к следующему элементу возможен только после доступа к предыдущему элементу.

Но оказывается можно совместить преимущества легкого управления списком и быстрый поиск, используя такие структуры данных, как *деревья*. Как еще один вариант, для быстрого поиска можно использовать *хеш-таблицы*. Но об этом можно узнать из других книг, например, из книги Дональда Кнута *Искусство программирования. Сортировка и поиск*. (*The Art of Computer Programming, Volume 3, Sorting and Searching* by Donald E. Knuth, Addison-Wesley). В ней обсуждаются типы данных, которые могут быть легко реализованы на языке программирования C с помощью тех приемов, которые уже описаны ранее.

Указатели и массивы

Одной из наиболее распространенных конструкций с использованием указателей являются массивы. Результатом использования указателей для массивов являются меньшее количество используемой памяти и высокая производительность. Причины, почему это происходит, и будут обсуждаться в этом разделе. Если вы создали массив с именем `values`, который содержит 100 целочисленных переменных, вы можете создать и указатель с именем `valuesPtr`, который будет использоваться для доступа к значениям элементов массива, с помощью следующего объявления.

```
int *valuesPtr;
```

Когда вы задаете указатель, который используется для доступа к элементам массива, вы обозначаете его не как “указатель на массив”, а как указатель на тот тип элементов, которые содержатся в массиве. Если вы имеете символьный массив с именем `text`, вы должны объявить указатель, который будет указывать на элементы массива, с помощью следующего утверждения.

```
char *textPtr;
```

Для того, чтобы указатель `valuesPtr` ссылался на первый элемент массива `values`, вы пишете следующее утверждение.

```
valuesPtr = values;
```

Оператор адресации в данном случае не используется, поскольку компилятор языка C трактует имя массива без индекса как указатель на массив. Поэтому после использования имени `values` без индекса будет получен указатель на первый элемент массива `values` (см. рис. 11.9).

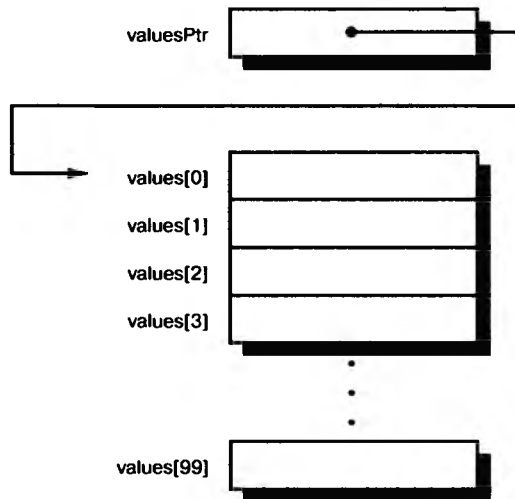


Рис. 11.9. Указатель на элементы массива

Соответственно, получить указатель на первый элемент массива `values` можно и с помощью оператора получения адреса, который используется совместно с первым элементом массива. Таким образом, утверждение:

```
valuesPtr = &values[0];
```

будет выполнять ту же роль, что и предыдущее утверждение и присваивать указателю `valuesPtr` значение, которое будет являться адресом первого элемента массива `values`.

Для того чтобы присвоить указателю `textPtr` значение, являющееся адресом первого элемента массива `text`, можно использовать следующее утверждение.

```
textPtr = text;
```

или

```
textPtr = &text[0];
```

В данном случае не важно, какое из утверждений вы выберете, это зависит от ваших предпочтений. Реальная польза от использования указателей получается при просмотре элементов массива. Если указателю `valuesPtr` был присвоен адрес первого элемента массива `values`, то выражение:

```
*valuesPtr
```

можно использовать для доступа к первому элементу массива `values`, который обычно обозначается `values[0]`. Для обращения к элементу массива `values[3]` с помощью указателя `valuesPtr`, вы должны добавить число 3 к указателю `valuesPtr` и затем использовать оператор разыменования.

```
*(valuesPtr + 3)
```

В общем случае используется выражение:

```
*(valuesPtr + i)
```

Поэтому установить значение 27 для элемента массива `values[10]` можно или с помощью очевидного утверждения

```
values[10] = 27;
```

или используя указатель `valuesPtr`, для чего необходимо написать следующее.

```
*(valuesPtr + 10) = 27;
```

Для того, чтобы заставить указатель `valuesPtr` ссылаться на второй элемент массива `values`, можно использовать оператор взятия адреса элемента `values[1]` и присвоить результат указателю `valuesPtr`:

```
valuesPtr = &values[1];
```

Если до этого указатель `valuesPtr` ссылался на первый элемент массива `values[0]`, то для того, чтобы он указывал на второй элемент массива, к нему можно просто добавить единицу.

```
valuesPtr += 1;
```

Это выражение полностью корректно в языке программирования C и может использоваться с указателями на любой тип данных. Поэтому в самом общем случае, если массив `a` содержит элементы типа `x`, а указатель `px` ссылается на элементы типа `x`, при этом переменные `i` и `n` являются целочисленными константами, то выполнение утверждения

```
px = a;
```

приведет к тому, что указатель `px` будет ссылаться на первый элемент массива `a`, а выражение

```
* (px + i)
```

будет ссылаться на значение, содержащееся в элементе `a[i]`. При этом утверждение

```
px += n;
```

заставит указатель `px` ссылаться на элемент массива, находящийся на расстоянии `n` от того, на который ранее указывал указатель, независимо от типа элементов, содержащихся в массиве.

При работе с указателями особенно удобны операторы инкремента “++” и декремента “--”. Использование оператора инкремента с указателем аналогично операции суммирования с единицей, а операция декремента имеет тот же эффект, что и вычитание единицы из указателя. Поэтому если переменная `textPtr` объявлена как указатель на тип `char` и при этом указывает на начальный элемент символьного массива с именем `text`, то выполнение утверждения

```
++textPtr;
```

приведет к тому, что указатель `textPtr` будет указывать на следующий символ массива `text[1]`. Аналогично, выполнение утверждения

```
--textPtr;
```

приведет к тому, что указатель `textPtr` будет ссылаться на предыдущий символ массива `text`. Разумеется, если предполагается, что до выполнения этого утверждения указатель `textPtr` не ссылался на начальный символ массива

В языке программирования C сравнение указателей является вполне корректной операцией. Это особенно полезно при сравнении двух указателей, ссылающихся на один и тот же массив. Например, вы производите проверку указателя `valuesPtr` для того, чтобы убедиться, что он не ссылается за последний элемент массива, состоящего из 100 элементов. При этом вы сравниваете его с указателем на последний элемент массива, для чего проверяете значение следующего выражения:

```
valuesPtr > &values[99]
```

которое будет иметь значение `TRUE` (не ноль), если указатель `valuesPtr` превышает адрес последнего элемента массива `values`, и значение `FALSE` (ноль) в противном случае. Вспомните предыдущее обсуждение и можете заменить это утверждение аналогичным. —

```
valuesPtr > values + 99
```

Поскольку переменная `values` используется без индекса, то она указывает на начальный элемент массива `values`. Вспомните, что это то же самое, что и запись `&values[0]`. В программе из листинга 11.11 демонстрируется использование указателей. Функция `arraySum` производит расчет суммы значений элементов, содержащихся в массиве.

Листинг 11.11. Работа с указателями в массиве

```
// Функция для суммирования элементов целочисленного массива.
#include <stdio.h>
int arraySum (int array[], const int n)
{
    int sum = 0, *ptr;
    int * const arrayEnd = array + n;

    for ( ptr = array; ptr < arrayEnd; ++ptr )
        sum += *ptr;

    return sum;
}

int main (void)
{
    int arraySum (int array[], const int n);
    int values[10] = { 3, 7, -9, 3, 6, -1, 7, 9, 1, -5 };
    printf ("The sum is %i\n", arraySum (values, 10));

    return 0;
}
```

Листинг 11.11. Вывод

The sum is 21

Внутри функции `arraySum` задается константный целочисленный указатель, который инициализируется адресом элемента, который находится непосредственно за последним элементом массива `array`. В цикле `for` производится последовательный просмотр элементов массива `array`. Значение указателя устанавливается таким, чтобы он ссылался на начальный элемент массива `array` в момент начала просмотра элементов массива. При каждой итерации элемент массива, на который ссылается указатель `ptr` добавляется к сумме элементов `sum`. Затем значение указателя `ptr` инкрементируется и он начинает ссылаться на следующий элемент массива `array`. Когда указатель будет ссылаться на последний элемент массива, выполнение цикла `for` заканчивается и значение переменной `sum` возвращается в вызывающую подпрограмму.

Оптимизация программ

На самом деле указатель, который указывает на элемент сразу за последним элементом массива, может быть удален, т.к. вы можете явно сравнивать значение указателя `ptr` и ссылку на этот элемент массива в заголовке цикла `for`:

```
for ( ...; pointer <= array + n; ... )
```

Единственным поводом для использования отдельного указателя на элемент массива сразу за последним элементом будет повышение производительности. Если записать цикл `for` так, как показано в последнем случае, то каждый раз при выполнении очередной итерации необходимо будет вычислять выражение `array+n`. Но поскольку выражение `array+n` никогда не изменяется в процессе выполнения цикла, то его можно заменить константой. Вычисляя его только один раз перед началом цикла вы будете

экономить время при каждой итерации за счет того, что исключите вычисление этого выражения.

Хотя в данном случае экономия будет совсем не заметна, т.к. используется всего 10 итераций при одном вызове функции `arraySum`. Но она будет довольно существенной, если неоднократно суммировать массивы большого размера.

Еще одним поводом для обсуждения оптимизации программ могут послужить сами указатели и их широкое использование в программе. Ранее уже обсуждалась функция `arraySum`, в которой использовался цикл `for` для суммирования элементов массива, обращение к которым производилось с помощью указателя. Но вполне можно было бы использовать индексы, и производить доступ к элементам массива с помощью индексации элементов массива `array[i]`, что более наглядно и понятно при чтении листингов. Но в общем случае процесс доступа к элементам массива по индексу занимает больше времени, чем использование указателей для этих целей. И именно это является основной причиной столь частого использования указателей в языке программирования C — генерируемый код будет более эффективным. Разумеется, если возникает необходимость производить нерегулярный, или непоследовательный доступ к элементам массива, т.е. использовать для целей доступа выражения `*(pointer+j)`, то заметного преимущества по времени выполнения может и не быть, т.к. это выражение выполняется так же долго, как и выражение `array[j]`.

Это массив или это указатель?

Вспомните, что для передачи массива в функцию, вы просто указываете имя массива, как вы делали, например, при вызове функции `arraySum`. Также вспомните, что для того, чтобы задать указатель на массив, вы использовали только имя массива. Это означает, что при вызове функции `arraySum` то, что было передано в функцию, в действительности является указателем на массив `values`. Именно поэтому вы могли изменять элементы массива из тела функции.

Но если это действительно тот случай, когда в функцию надо передавать указатель на массив, то может вызвать удивление тот факт, что формальный параметр в заголовке функции объявлен не как указатель, т.е. не так, как показано ниже.

```
int *array;
```

Должны ли все ссылки на массив в теле функции производиться с помощью указателей?

Для ответа на этот вопрос вспомним обсуждение темы об указателях и массивах. Как уже упоминалось, если указатель `valuesPtr` ссылается на тот тип данных, которые являются элементами массива `values`, то выражение `*(valuesPtr+i)` полностью эквивалентно выражению `values[i]` с учетом того, что указателю `valuesPtr` присвоен адрес начального элемента массива. Из этого следует, что всегда можно использовать выражение `*(valuesPtr+i)` для доступа к элементу массива `values` с индексом `i`, а в общем случае, если `x` является массивом любого типа, то в языке программирования C выражение `*(x+i)` всегда можно сделать эквивалентным выражению `x[i]`.

То есть хорошо видно, что указатели и массивы в языке C находятся в довольно близких отношениях, и именно поэтому в функции `arraySum` можно объявлять массив с именем `array` как тип “массив целых чисел” или как тип “указатель на целое число”. Любое из этих объявлений будет вполне корректным, в чем легко можно убедиться.

Если вы будете использовать индексы для доступа к элементам массива, то при объявлении функции используйте формальный параметр как тип “массив целых чисел”. Это более наглядно отображает способ обращения к элементам массива в теле

функции. И наоборот, если вы используете аргумент как указатель, то и параметр объявите как указатель.

Итак представьте, что в предыдущем примере программы вы объявили массив `array` как указатель на целое число и затем использовали его для последовательного доступа к элементам массива. При этом вы можете исключить переменную `ptr` и использовать вместо нее переменную `array`, что и сделано в программе из листинга 11.12.

Листинг 11.12. Суммирование элементов массива

```
// Функция суммирования элементов целочисленного массива. Версия 2.
#include <stdio.h>
int arraySum (int *array, const int n)
{
    int sum = 0;
    int * const arrayEnd = array + n;

    for ( ; array < arrayEnd; ++array )
        sum += *array;

    return sum;
}

int main (void)
{
    int arraySum (int *array, const int n);
    int values[10] = { 3, 7, -9, 3, 6, -1, 7, 9, 1, -5 };
    printf ("The sum is %i\n", arraySum (values, 10));

    return 0;
}
```

Листинг 11.12. Вывод

```
The sum is 21
```

Эта программа довольно легкая для понимания. Первое выражение в заголовке цикла `for` пропущено, поскольку ни одно из значений не инициализируется до того, как цикл начнет выполняться. Еще раз рассмотрим тот момент, когда вызывается функция `arraySum`. При этом в функцию передается указатель на массив `values`, который и используется в теле функции. Изменения значений указателя `array` (в отличие от значений, на которые он указывает) не оказывает никакого влияния на содержимое массива `values`. Потому оператор инкремента, который используется с указателем `array`, изменяет только значения самого указателя и не изменяет содержимое массива `values`. (Разумеется, вы можете изменять содержимое массива, если это необходимо, для чего используйте оператор разыменовывания.)

Указатели на символьные строки

Наиболее часто указатели используются с символьными массивами по причине удобства записи и эффективности. Для того чтобы продемонстрировать легкость использования указателей с символьными массивами, напомним функцию `copyString`, которая производит копирование одной строки в другую. Если написать эту функцию с использованием обычных методов индексации, то описание функции может выглядеть следующим образом.

```

void copyString (char to[], char from[])
{
    int i;
    for ( i = 0; from[i] != '\0'; ++i )
        to[i] = from[i];
    to[i] = '\0';
}

```

Здесь цикл `for` выполняется до тех пор, пока не встретится нулевой символ, который не будет скопирован в выходную строку, чем и объясняется наличие второго утверждения в теле функции. Если же закодировать функцию `copyString` с помощью указателей, то отпадает необходимость использования индексов. Версия с использованием указателей показана в программе из листинга 11.13.

Листинг 11.13. Версия функции `copyString` с указателями

```

#include <stdio.h>
void copyString (char *to, char *from)
{
    for ( ; *from != '\0'; ++from, ++to )
        *to = *from;
    *to = '\0';
}

int main (void)
{
    void copyString (char *to, char *from);
    char string1[] = "A string to be copied.";
    char string2[50];
    copyString (string2, string1);
    printf ("%s\n", string2);
    copyString (string2, "So is this.");
    printf ("%s\n", string2);

    return 0;
}

```

Листинг 11.13. Вывод

```

A string to be copied.
So is this.

```

В функции `copyString` используются два формальных параметра: указатель на символы `to` и `from`, а не на символьные массивы, как это было сделано в предыдущей версии функции `copyString`. Это точнее отражает суть использования этих переменных в функции.

Затем для копирования строк используется цикл `for` (без инициализации исходных переменных) для копирования строки, на которую ссылается указатель `from` в строку, на которую ссылается указатель `to`. При каждой итерации указатели `from` и `to` инкрементируются на единицу. Это приводит к тому, что указатель `from` будет ссылаться на очередной символ, который необходимо копировать, а указатель `to` будет указывать на место непосредственно за последним скопированным символом в выходной строке.

Когда значение, на которое ссылается указатель `from`, будет представлять нулевой символ, цикл `for` заканчивается. Затем в конец выходной строки помещается нулевой символ, который определяет окончание строки.

В подпрограмме `main` функция `copyString` вызывается дважды, первый раз для копирования содержимого строки `string1` в строку `string2`, и второй раз — для копирования содержимого строковой константы `"So is this."` в строку `string2`.

Строковые символьные константы и указатели

На самом деле из корректного вызова функции в предыдущей программе

```
copyString (string2, "So is this.");
```

можно понять, что в данном случае строковая символьная константа, которая передается в функцию в качестве аргумента, реально преобразовывается в указатель на символьную строку.

Это работает не только в таких случаях и преобразование строковой константы в указатель можно обобщить на все ситуации, где в языке программирования C используются символьные строки. Поэтому, если переменная объявлена как указатель на символ, т.е.:

```
char *textPtr;
```

то выполнение утверждения:

```
textPtr = "A character string.";
```

приводит к тому, что указатель `textPtr` будет ссылаться на строковую символьную константу `"A character string."`. Но будьте внимательны и не путайте указатели на символы и указатели на массивы символов, с которыми присваивание, которое было только что использовано, не допустимо. Поэтому, например, если переменную `text` объявить как массив символов

```
char text[80];
```

то уже нельзя будет написать следующее утверждение.

```
text = "This is not valid.";
```

Единственный случай, когда это можно сделать, это только при инициализации массива во время объявления массива символов:

```
char text[80] = "This is okay.";
```

Такая инициализация массива не приводит к сохранению указателя на символьную строку `"This is okay."` внутри переменной `text`, а сохраняет только сами символы в соответствующих элементах массива `text`.

Если переменная `text` будет указателем на тип `char`, инициализация этой переменной следующим образом

```
char *text = "This is okay.";
```

присвоит ей указатель на символьную строку `"This is okay."`

Другой пример различия между символьными строками и указателями на символьные строки показан в инициализации символьного массива `days`, который содержит указатели на названия дней недели.

```
char *days[] =
{ "Sunday", "Monday", "Tuesday", "Wednesday", "Thursday",
  "Friday", "Saturday" };
```

Массив `days` объявлен для хранения семи элементов, каждый из которых является указателем на символьную строку. Поэтому элемент `days[0]` содержит указатель на строку "Sunday", элемент `days[1]` содержит указатель на строку "Monday" и т.д. (см. рис. 11.10). Вы можете отобразить название третьего дня недели с помощью следующего утверждения.

```
printf ("%s\n", days[3]);
```

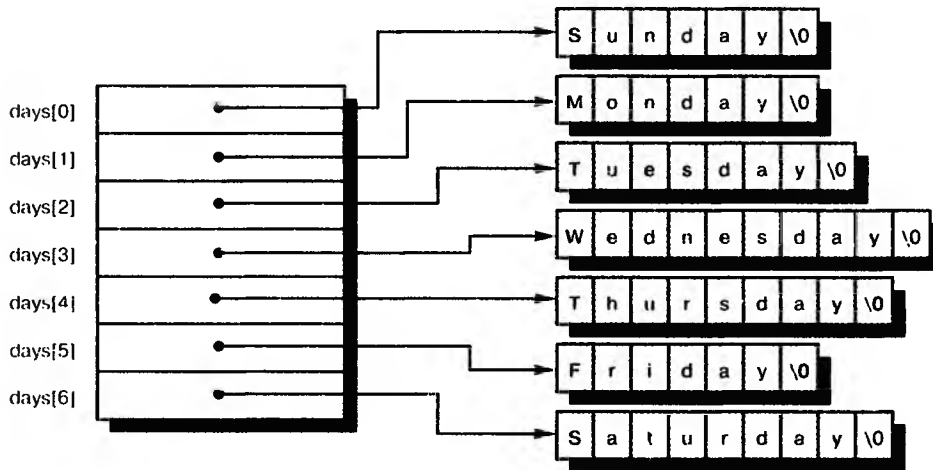


Рис. 11.10. Массив указателей

Еще об операторах инкремента и декремента

До этого момента, где бы вы ни использовали операторы инкремента и декремента, вы использовали их как отдельные выражения. Когда вы писали выражение `++x`, вы знали, что таким образом вы добавляете 1 к переменной `x`. И, как вы уже неоднократно видели, если переменная `x` является указателем на массив, то ее использование совместно с оператором инкремента приводит к изменению адреса, по которому производится ссылка, и обращение будет происходить к следующему элементу массива.

Операторы инкремента и декремента могут использоваться в выражениях, в которых есть и другие операторы. При этом необходимо точно знать, как эти операторы будут себя вести.

До сих пор при использовании операторов инкремента и декремента вы всегда помещали их перед переменной, которую собирались инкрементировать или декрементировать. Для инкремента переменной `i` вы просто писали следующее.

```
++i;
```

На самом деле будет вполне корректно помещать оператор инкремента и после переменной, как это сделано в следующем утверждении.

```
i++;
```

Оба выражения полностью корректны и оба выполняют одно и то же, а именно инкрементируют значение переменной *i*. В первом случае, когда оператор “++” помещен перед операндом, операцию инкремента точнее можно назвать *преинкремент*. Во втором случае, когда оператор “++” находится после операнда, операцию следует называть *постинкремент*.

Аналогичные выводы можно сделать и для оператора декремента. Поэтому утверждение

```
--i;
```

выполняет операцию *предекремента* для переменной *i*, а утверждение:

```
i--;
```

выполняет *постдекремент* для переменной *i*. В обоих случаях результатом будет вычитание единицы из значения переменной *i*.

Но что происходит с операциями инкремента и декремента когда они используются в более сложных выражениях, когда различие между приставками “пре” и “пост” будет использоваться при вычислении? Предположим, вы задали две целочисленные переменные *i* и *j*. Если вы присвоите переменной *i* значение 0 и затем напишите утверждение

```
j = ++i;
```

то значение, которое будет присвоено переменной *j* будет 1, а не 0, как можно было ожидать. Так как в этом случае используется операция преинкремента, то переменная инкрементируется до того, как это значение будет использоваться в выражении.

Таким образом, в предыдущем утверждении значение переменной *i* будет сначала инкрементировано и примет значение 1, и только после этого данное значение будет присвоено переменной *j*, как это можно сделать с помощью следующих двух утверждений.

```
++i;  
j = i;
```

Но если использовать операцию постинкремента, как показано в следующем утверждении

```
j = i++;
```

то переменная *i* будет инкрементироваться после того, как ее значение будет присвоено переменной *j*. Таким образом, если значение переменной *i* равно 0 до того, как будет выполнено предыдущее утверждение, то сначала переменной *j* будет присвоено значение 0 и только после этого переменная *i* будет инкрементирована, как это можно сделать с помощью следующих утверждений

```
j = i;  
++i;
```

которые можно использовать вместо одного предыдущего утверждения. Еще один пример, если переменная *i* равна 1, то утверждение

```
x = a[--i];
```

выполнит присваивание значения элемента множества *a[0]* переменной *x*, поскольку переменная *i* будет декрементирована до того, как ее значение будет использовано при поиске значения по индексу в массиве *a*. Утверждение

```
x = a[i--];
```

в котором используется операция постдекремента, присвоит переменной *x* значение элемента множества *a[1]*, поскольку переменная *i* будет декрементирована после того, как ее значение будет использовано в качестве индекса.

В третьем примере, показывающем различие между операциями пре- и постинкрементации, используется вызов функции

```
printf ("%i\n", ++i);
```

где сначала проводится инкрементация переменной *i* и только потом ее значение используется в функции `printf`, тогда как при таком вызове функции

```
printf ("%i\n", i++);
```

инкремент переменной *i* будет выполнен только после того, как ее значение будет использовано в функции. Поэтому если переменная *i* содержит значение 100, то при первом вызове функции `printf` будет отображено значение 101, а при втором вызове функции `printf` отобразится значение 100. В этом случае переменная *i* станет равной 101 только после того, как выполнится функция `printf`.

И наконец, рассмотрим последний пример перед тем, как представить программу из листинга 11.14. Если переменная *textPtr* является указателем, то выражение

```
* (++textPtr)
```

сначала инкрементирует значение переменной *textPtr* и только потом будет выбран символ, на который она указывает, в то время как выражение

```
*(textPtr++)
```

сначала выберет символ, на который указывает переменная *textPtr*, и только потом ее значение будет инкрементировано. В любом случае круглые скобки можно не ставить, т.к. операторы "*" и "++" имеют одинаковый приоритет, но выражение выполняется справа налево.

Теперь вернемся к функции `copyString` из листинга 11.13 и перепишем ее с использованием операций инкремента непосредственно в утверждениях присваивания.

Поскольку указатели *to* и *from* инкрементируются каждый раз после утверждения присваивания внутри цикла `for`, они должны быть преобразованы как постинкрементные операции. Отредактированный цикл `for` для программы из листинга 11.13 теперь будет выглядеть так.

```
for ( ; *from != '\0'; )  
    *to++ = *from++;
```

Выполнение утверждения присваивания внутри цикла происходит следующим образом. Символ, на который ссылается указатель *from*, извлекается и после этого инкрементируется указатель *from*, указывая на следующий символ в исходной строке. Этот символ затем сохраняется в выходной строке в том месте, на которое ссылается указатель *to*, после чего указатель *to* инкрементируется и начинает указывать на следующее место в выходной строке, куда должен быть помещен очередной символ.

Изучайте предыдущее утверждение до тех пор, пока вы полностью не будете понимать все операции. Утверждения такого типа настолько часто используются в языке программирования C, что очень важно их понимать, прежде чем двигаться дальше.

Предыдущий цикл `for` выглядит несколько необычно, поскольку он не имеет выражений ни для инициализации переменных цикла, ни для изменения этих

переменных. Логика цикла будет более понятна, если использовать цикл `while`. Именно это и сделано в программе из листинга 11.14. В этой программе представлена новая версия функции `copyString`. В цикле `while` используется тот факт, что значением нулевого символа является 0, что обычно и используется программистами на языке C.

Листинг 11.14. Улучшенная версия функции `copyString`

```
// Функция копирования одной строки и в другую. Указатели. Версия 2.
#include <stdio.h>
void copyString (char *to, char *from)
{
    while ( *from )
        *to++ = *from++;
    *to = '\0';
}
int main (void)
{
    void copyString (char *to, char *from);
    char string1[] = "A string to be copied.";
    char string2[50];

    copyString (string2, string1);
    printf ("%s\n", string2);
    copyString (string2, "So is this.");
    printf ("%s\n", string2);

    return 0;
}
```

Листинг 11.14. Вывод

```
A string to be copied.
So is this.
```

Операции с указателями

Как вы уже успели убедиться, можно легко добавлять и вычитать целые числа из указателей. Более того, вы можете сравнить два указателя и определить их равенство или то, что один указатель больше другого. Еще одной операцией, которую можно производить над указателями, является вычитание одного указателя из другого. В результате вычитания двух указателей получается число элементов, расположенных между двумя указателями.

Поэтому если `a` указывает на массив элементов определенного типа, а `b` указывает на какой либо элемент из этого же массива, то выражение `b-a` будет содержать число элементов между этими двумя указателями. На пример, если `p` указывает на элемент массива `x`, то в утверждении

$$n = p - x;$$

переменной `n` (подразумевается, что это переменная целочисленного типа) будет присвоен индекса элемента внутри массива `x`, на который ссылается указатель `p`. (Необходимо отметить, что в языке программирования C тип, который получается

при вычитании двух указателей, называется `ptrdiff_t`, и он определен в заголовочном файле `stddef.h`.)

Таким образом, если `p` будет указывать на сотый элемент массива `x`, что можно сделать с помощью утверждения

```
p = &x[99];
```

то значение `n` после предыдущего вычитания будет равно 99.

Как практическое применение только что рассмотренного способа вычитания указателей, познакомьтесь в новой версии функции `strlen` из главы 10. В программе из листинга 11.15 указатель на символ `cptr` используется для последовательного просмотра символов строки `string`, пока не будет достигнут нулевой символ. После чего указатель `string` вычитается из указателя `cptr` для получения числа элементов (символов), содержащихся в строке. Из вывода, сделанного программой, можно убедиться, что программа работает правильно.

Листинг 11.15. Использование указателей для определения длины строки

```
// Функция для определения длины строки - версия с указателями.
#include <stdio.h>
int strlen (const char *string)
{
    const char *cptr = string;
    while ( *cptr )
        ++cptr;
    return cptr - string;
}

int main (void)
{
    int strlen (const char *string);

    printf ("%i ", strlen ("strlen test"));
    printf ("%i ", strlen (""));
    printf ("%i\n", strlen ("complete"));

    return 0;
}
```

Листинг 11.15. Вывод

```
17 0 8
```

Указатели на функции

В следующих разделах коснемся более сложных вопросов использования указателей и сейчас рассмотрим указатели на функции. Когда в языке C производится работа с указателями на функции, то компилятору необходимо знать не только то, что указатель ссылается на функцию, но и тип возвращаемого значения, и число и типы параметров.

Для того, чтобы объявить переменную `fnPtr` типа "указатель на функцию, которая возвращает значение типа `int` и не имеет параметров", необходимо сделать следующее объявление.

```
int (*fnPtr) (void);
```

Круглые скобки вокруг выражения `*fnPtr` необходимы, иначе компилятор языка C будет трактовать это утверждение как объявление функции `fnPtr`, которая возвращает указатель на целое число, поскольку оператор вызова функции `()` имеет более высокий приоритет, чем оператор разыменовывания `*`.

Для того чтобы заставить указатель указывать на заданную функцию, следует просто присвоить ему имя функции. Таким образом, если идентификатор `lookup` является именем функции, которая возвращает целочисленное значение и не имеет аргументов, то утверждение

```
fnPtr = lookup;
```

сохранит ссылку на эту функцию в указателе на функцию `fnPtr`. Запись имени функции без последующих круглых скобок имеет тот же эффект, как и запись имени массива без индекса. При этом компилятор языка C автоматически создаст указатель на заданную функцию, а значит нет необходимости ставить перед именем функции символ амперсанта `&`.

Если функция `lookup` предварительно не была объявлена в программе, то это необходимо сделать до того, как эта функция будет использоваться. Поэтому утверждение, наподобие

```
int lookup (void);
```

необходимо поместить в программе до того, как эта функция будет присваиваться переменной `fnPtr`. После чего вы можете вызывать функцию косвенно, с помощью указателя на функцию, используя при этом оператор вызова функции `()` и добавляя необходимые аргументы, если они требуются. Например, утверждение

```
entry = fnPtr ();
```

приведет к вызову функции, на которую ссылается указатель `fnPtr`, и сохранит возвращаемое функцией число в переменной `entry`.

Наиболее общим случаем использования указателей на функции является передача их в качестве аргумента в другие функции. В стандартных библиотеках языка C это применяется, например, в функции `qsort`, которая выполняет "быструю сортировку" элементов массивов данных. Одним из аргументов этой функции является указатель на функцию, когда функции `qsort` необходимо сравнить два элемента сортируемого массива. Поэтому функция `qsort` может использоваться для сортировки массивов любого типа, т.к. сравнение элементов массива выполняется пользователем в специально разрабатываемой функции, а не внутри самой функции `qsort`. В приложение Б, "Стандарт библиотеки языка программирования C" содержатся дополнительные детали о функции `qsort` и приводятся примеры ее использования.

Другим наиболее общим случаем использования указателей на функции является создание так называемых "таблиц диспетчеризации". Нельзя создать массив, состоящий из функций, однако вполне допустимо сохранять указатели на функции внутри массива. Благодаря этому, можно создавать таблицы указателей на функции, которые необходимо вызывать. Например, вы можете создать таблицу для выполнения различных команд, вводимых пользователем. Каждый элемент таблицы будет содержать как имя команды, так и указатель на функцию, которая должна выполнять эту команду. Теперь, когда пользователь вводит отдельную команду, необходимо только посмотреть в таблицу и вызвать соответствующую функцию для выполнения этой команды.

Указатели и адреса памяти

Перед окончанием обсуждения темы указателей в языке программирования C необходимо прояснить детали, как это физически реализовано. Память компьютера можно рассматривать как последовательность ячеек для хранения данных. Каждая ячейка памяти имеет связанный с ней номер, который называется *адресом*. Обычно первая ячейка памяти имеет номер 0. В большинстве компьютерных систем ячейку памяти называют *байтом*.

Компьютер использует память для хранения команд вашей программы и хранения данных для всех переменных, связанных с программой. Поэтому если вы объявите переменную `count`, которая будет иметь тип `int`, система выделит место в памяти для хранения значения переменной `count` при выполнении программы. Это может быть, например, ячейка с номером 500.

При этом одно из преимуществ языков высокого уровня, подобных языку C, заключается в том, что вам не нужно самим распределять память для переменных, все это будет сделано за вас автоматически. Но понимание того, что с каждой переменной связана некоторая ячейка памяти поможет вам понять операции с указателями.

Когда вы используете оператор взятия адреса для переменной, число, которое вы получаете, является адресом ячейки памяти, где хранится значение этой переменной. Поэтому утверждение

```
intPtr = &count;
```

присвоит переменной `intPtr` адрес ячейки памяти, который связан с переменной `count`. Поэтому если значение переменной размещено по адресу 500 и представляет число 10, то это утверждение присвоит число 500 переменной `intPtr`, как показано на рис. 11.11.

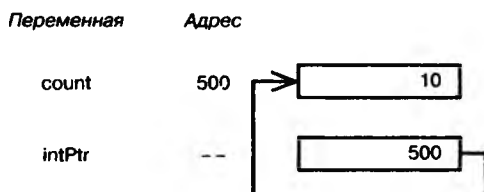


Рис. 11.11. Указатели и адреса памяти

Адрес, где хранится значение переменной `intPtr`, в данном случае не важен.

Использование оператора разыменовывания совместно с указателем, как показано в выражении:

```
*intPtr
```

приведет к извлечению значения, хранящегося по адресу, который представляет значение указателя. Это значение извлекается и интерпретируется в соответствии с типом, который использовался при объявлении указателя. Таким образом, если указатель `intPtr` должен ссылаться на тип `int`, то значение, хранящееся по данному адресу в памяти, для обращения к которому используется выражение

```
*intPtr
```

интерпретируется системой как целое число. В результате будет получено значение 10, которое является целым числом.

Для сохранения значения в памяти, адресом которого является значение указателя, используется утверждение

```
*intPtr = 20;
```

при выполнении которого совершаются аналогичные действия. Содержимое указателя `intPtr` трактуется как адрес памяти. Затем заданное значение помещается по данному адресу. В нашем примере целое число 20 будет сохранено в памяти по адресу 500.

Иногда системным программистам необходимо знать размещение отдельных значений в ячейках памяти. Поэтому понимание процесса работы с памятью будет отнюдь не лишним.

Как вы могли убедиться, прочитав эту главу, указатели являются очень мощным средством языка программирования C. В этой главе было продемонстрировано довольно много способов применения указателей. Но это далеко не все и, например, вы можете задать указатель на указатель, или указатель на указатель на указатель. Такие сложные конструкции не рассматриваются в этой книге, хотя они являются простым логическим расширением того, что уже описано в этой главе. Но покорять логические вершины построения указателей для новичка довольно сложная задача. Рекомендую еще раз перечитать все разделы этой главы и добиться полного понимания работы указателей. Решение упражнений, которые приведены ниже, поможет закрепить изученный материал.

Упражнения

1. Наберите и выполните все 15 программ, представленных в этой главе. Сравните полученные вами результаты с теми, что приведены в выводах для каждой программы из книги.
2. Напишите функцию `insertEntry`, которая должна вставлять новый элемент в связанный список. В качестве аргументов должны использоваться указатель на связанный список и указатель на элемент списка, после которого должен быть вставлен новый элемент.
3. Функция, разработанная в упражнении 2, будет вставлять элемент после уже существующего элемента списка, поэтому нет возможности вставить элемент в начало списка. Используйте ту же функцию и доработайте ее так, чтобы можно было вставлять элемент в начало списка. (Подсказка: можно создать специальную структуру, указывающую на начало списка.)
4. Напишите функцию `removeEntry` для удаления элемента связанного списка. Единственный аргумент этой функции должен быть указателем на список. Функция должна удалять тот элемент, который находится сразу после указываемого элемента. (Подумайте, почему вы не можете удалить тот элемент, на который ссылается указатель.) Вы должны использовать специальную структуру, о которой упоминалось в упражнении 3, чтобы удалить первый элемент списка.
5. Двойным связанным списком называется список, в котором каждый элемент содержит не только указатель на следующий элемент, но и указатель на предыдущий элемент списка. Задайте соответствующую структуру для элемента двойного связанного списка и затем напишите небольшую программу, которая реализует двойной связанный список и выводит значения всех его элементов.

6. Разработайте функции `insertEntry` и `removeEntry` для двойного связанного списка, аналогично функциям, которые вы разрабатывали в предыдущих упражнениях. Почему в данном случае вы можете использовать указатель на непосредственно удаляемый элемент?
7. Напишите версию с указателями для функции `sort` из главы 8, "Функции", причем так, чтобы указатели использовались по всем случаям, включая индексацию переменных в цикле.
8. Напишите функцию `sort3` для сортировки трех целых чисел в возрастающем порядке. Не используйте при этом массивы.
9. Перепишите функцию `readLine` из главы 10 таким образом, чтобы в ней использовался указатель на символ вместо массива.
10. Перепишите функцию `compareStrings` из главы 10 таким образом, чтобы в ней использовались указатели на символ вместо массивов.
11. Используя описание структуры `date`, которое приводится в данной главе, напишите функцию `dateUpdate`, которая принимает указатель на структуру `date` и изменяет значение дня на следующее (см. листинг 9.4).
12. Используя следующие объявления:

```
char *message = "Programming in C is fun\n";
char message2[] = "You said it\n";
char *format = "x = %i\n";
int x = 100;
```

определите, какие наборы утверждений `printf` будут выполнены правильно.

```
/** набор 1 */
printf ("Programming in C is fun\n");
printf ("%s", "Programming in C is fun\n");
printf ("%s", message);
printf (message);
/** набор 2 */
printf ("You said it\n");
printf ("%s", message2);
printf (message2);
printf ("%s", &message2[0]);
/** набор 3 */
printf ("said it\n");
printf (message2 + 4);
printf ("%s", message2 + 4);
printf ("%s", &message2[4]);
/** набор 4 */
printf ("x = %i\n", x);
printf (format, x);
```


Операции с битами

Как уже упоминалось, язык программирования C создавался для разработки приложений, работающих на системном уровне. Указатели именно для того и существуют, чтобы предоставить программисту абсолютную степень контроля и доступа к памяти компьютера. Также аналогично указателям, системные программисты часто работают с битами в отдельных словах. Язык программирования C предоставляет несколько операций, специально созданных для работы с отдельными битами.

Вспомним обсуждение понятия *байта* в предыдущих главах. В большинстве компьютерных систем, байт состоит из восьми неделимых частей, называемых *битами*. Бит может принимать только два значения: 1 или 0. Поэтому байт, сохраненный по адресу 100 в памяти компьютера, может рассматриваться как строка из восьми бинарных единиц, представленная следующим образом.

01100100

Крайний правый бит в байте называется *наименьшим значащим битом* или *младшим битом*, а крайний левый бит называется *наибольшим значащим битом*, или *старшим битом*. Если трактовать строку битов как целое число, то самый младший бит можно представить как число 2^0 , или 1, бит правее его будет представлять число 2^1 , или 2, еще правее бит будет числом 2^2 , или 4 и т.д. Поэтому предыдущее двоичное число можно представить как десятичное число $2^2+2^5+2^6=4+32+64=100$.

Представление отрицательных чисел выполняется несколько по-другому. В большинстве компьютеров для таких чисел используется *двоичное дополнение*, или дополнение до двух. При таком представлении числа кратный левый бит является знаковым битом. Если этот бит равен единице, то число считается отрицательным, а если бит равен нулю, то число положительное. Оставшаяся часть бита будет представлять значение числа. При двоичном дополнении, значение -1 будет выражено байтом, в котором все биты установлены в 1.

11111111

Удобно преобразовывать отрицательные десятичные числа в двоичные следующим способом. Сначала к отрицательному числу добавляется единица и абсолютное значение нового числа выражается в двоичном виде. В получившемся двоичном числе инвертируются все биты, т.е. 0 становится 1, а 1 становится 0. Это и будет двоичным аналогом отрицательного десятичного числа. Например, преобразуем число -5 в двоичное, для чего прибавим к нему 1 и получим -4 . Абсолютное значение 4 выражаем в

двоичном виде, что будет 00000100, и инвертируем все биты. Окончательным результатом будет 11111011.

Для преобразования отрицательного двоичного числа в десятичное необходимо проделать обратные действия, т.е. инвертировать все биты, преобразовать результат в десятичное значение, изменить знак результата и вычесть единицу.

Заканчивая обсуждение двоичного дополнения, отметим, что наибольшим положительным числом, которое может быть сохранено в n битах, будет $2^{n-1}-1$. Для восьми битов можно сохранить значение $2^{7-1}-1$, или 127. Аналогично, наименьшим отрицательным числом, которое может быть сохранено в n битах, будет -2^{n-1} , что для восьми битов будет соответствовать числу -128 . (Сможете подсчитать, почему абсолютные величины положительных и отрицательных значений не совпадают?)

В большинстве современных компьютеров целые числа занимают четыре последовательных байта, или 32 бита компьютерной памяти. Поэтому наибольшим положительным числом, которое может быть сохранено в памяти, будет $2^{31}-1$, или 2 147 483 647, а наименьшим отрицательным числом является число -2^{31} , или -2 147 483 648. В главе 4, “Переменные, типы данных и арифметические выражения”, вы познакомились с типом `unsigned`, с помощью которого можно эффективно повысить максимальное значение переменной. Это происходит потому, что самый старший бит больше не используется как знаковый бит и именно этот “экстра”-бит помогает увеличить диапазон положительных чисел вдвое. Если быть точным, то заданное количество битов n может использоваться для хранения 2^n-1 значений. В машинах, использующих слова размером 32 бита, диапазон беззнаковых целых чисел может простирается от 0 до 4 294 967 296.

Операции с битами

Теперь, когда вы уже имеете начальные сведения о битах, обсудим различные операции, которые можно производить над отдельными битами. Операторы, которые используются в языке C для манипуляции с битами, представлены в табл. 12.1.

Таблица 12.1. Операторы для работы с битами

Символ	Операция
&	Поразрядное И
	Поразрядное включающее ИЛИ
^	Поразрядное исключающее ИЛИ
~	Поразрядная инверсия
<<	Сдвиг влево
>>	Сдвиг вправо

Все операторы, за исключением оператора “~”, являются бинарными операторами и используются с двумя операндами. Операции с битами в языке C могут производиться только над целочисленными типами, такими как `short`, `long`, `long long` и `signed` или `unsigned`. Также эти операции могут выполняться и с символами, но не с вещественными числами.

Поразрядный оператор &

Когда над двумя значениям производится операция поразрядного И (&), двоичные представления чисел сравниваются бит за битом. Для каждого бита, который имеет значение 1 в первом числе, и соответствующего бита второго числа, также имеющего значение 1, результирующий бит будет иметь значение 1. В любом другом случае результирующее значение будет равно 0. Если биты b1 и b2 являются соответствующими битами двух операндов, то следующая таблица, называемая *таблицей истинности*, отображает результат операции И для всех возможных значений b1 и b2.

b1	b2	b1 & b2
0	0	0
0	1	0
1	0	0
1	1	1

Например, если переменные w1 и w2 заданы как тип short ints и значение w1 равно 25, а значение w2 равно 77, то в утверждении

```
w3 = w1 & w2;
```

переменной w3 будет присвоено значение 9. Это легко понять, если представить значения переменных как двоичные числа. Предположим, что тип short int имеет размер 16 битов. При этом получим следующее.

```
w1 0000000000011001 (25)
w2 0000000001001101 & (77)
-----
w3 000000000001001 ( 9)
```

Когда вы начнете разбираться в работе логического оператора И "&&", который возвращает значение "истина", если только оба операнда имеют значение "истина", вы вспомните работу поразрядного оператора И "&" и заметите много общего. К сожалению, начинающие программисты часто путают эти два оператора.

Логический оператор И "&&" используется в логических выражениях для получения результирующего значения "ложь" или "истина" и не выполняет поразрядного сравнения.

Поразрядный оператор И "&" часто используется для операций маскирования. Используя этот оператор, можно легко установить заданный бит данных в значение 0. Например, в утверждении

```
w3 = w1 & 3;
```

переменной w3 присваивается значение переменной w1, для которой выполнена поразрядная операция И со значением 3. При этом все биты левее двух младших битов устанавливаются в 0, а остальные биты сохраняют свое значение.

Подобно всем бинарным арифметическим операторам языка C, бинарные операторы для битов можно использовать со знаком оператора присваивания, получая более компактную запись. Таким образом, утверждение

```
word &= 15;
```

выполняет то же самое, что и утверждение

```
word = word & 15;
```

и это соответствует установке четырех правых битов переменной `word` и установки остальных бит в значение 0.

При использовании констант в поразрядных операциях обычно их удобно выражать в восьмеричном или шестнадцатеричном виде. Выбор того или иного вида зависит от размера данных, с которыми производится работа. Например, при использовании 32-разрядного компьютера удобно использовать шестнадцатеричную нотацию, поскольку 32 делится без остатка на 4.

В программе из листинга 12.1 демонстрируется работа поразрядного оператора **И**. Поскольку для демонстрации удобнее использовать только положительные числа, все переменные имеют тип `unsigned int`

Листинг 12.1. Поразрядный оператор И

```
// Демонстрация работы поразрядного оператора И
#include <stdio.h>
int main (void)
{
    unsigned int word1 = 077u, word2 = 0150u, word3 = 0210u;

    printf ("%o ", word1 & word2);
    printf ("%o ", word1 & word1);
    printf ("%o", word1 & word2 & word3);
    printf ("%o\n", word1 & 1);

    return 0;
}
```

Листинг 12.1. Вывод

```
50 77 10 1
```

Вспомните, что если целая константа начинается с нуля, то она представлена в восьмеричном виде (основанием является число 8). Поэтому три целочисленные переменные `word1`, `word2` и `word3` типа `unsigned int` имеют значения 077, 0150 и 0210 соответственно. Вспомните также, что в главе 4 говорилось о том, что если непосредственно за константой следует символ `u`, то эта константа трактуется как беззнаковая.

При первом вызове функции `printf` будет отображено число 50, как результат поразрядной операции **И** над операндами `word1` и `word2`. Ниже объясняется, как получается этот результат.

<code>word1 ...</code>	<code>000 111 111</code>	<code>(077)</code>
<code>word2 ...</code>	<code>001 101 000</code>	<code>& (0150)</code>

<code>...</code>	<code>000 101 000</code>	<code>(050)</code>

Здесь приведены только первые девять байтов справа для всех значений, поскольку все остальные биты в результате операции будут установлены в 0. Двоичные представления чисел разбиты на группы по три бита для более удобного преобразования чисел между двоичным и восьмеричным представлениями.

При втором вызове функции `printf` отображается восьмеричное число 77, которое является результатом применения поразрядной операции **И** к операнду `word1`, используемому дважды. По определению поразрядная операция **И**, выполненная над одним и тем же значением `x`, в результате возвратит `x`.

При третьем вызове функции `printf` отобразится результат выполнения поразрядной операции **И** над тремя операндами `word1`, `word2` и `word3`. При последовательном использовании оператора поразрядного **И** можно не учитывать порядок выполнения операций, т.к. выражение `a&b&c` может выполняться и как `(a&b) &c`, и как `a &(b&c)`, но считается, что выполнение производится слева направо. В качестве упражнения, вы можете проверить результат выполнения и убедиться, что при последовательном выполнении операции **И** над словами `word1`, `word2` и `word3`, получится восьмеричное число 10.

При последнем вызове функции `printf` производится отображение значения самого младшего бита слова `word1`. Это можно использовать для определения того, является ли целое число четным или нечетным, поскольку у нечетных чисел младший бит всегда будет равен 1, а у четных — всегда 0. Поэтому при выполнении выражения

```
if ( word1 & 1 )
```

будет получен результат `true`, если значение переменной `word1` является нечетным, и `false`, если число четное.

Поразрядный оператор |

Когда над двумя значения производится операция поразрядного **ИЛИ** (`|`), то последовательно сравниваются значения всех битов при двоичном представлении этих значений. Если при этом соответствующий бит имеет значение 1 в первом или втором операнде, то результирующее значение будет равно 1. Таблица истинности для операции неисключающее **ИЛИ** приведена ниже.

b1	b2	b1 b2
0	0	0
0	1	1
1	0	1
1	1	1

Таким образом, если операнд `w1` типа `unsigned int` имеет восьмеричное значение 0431, а операнд `w2` типа `unsigned int` равен 0152, то в результате применения операции поразрядного **ИЛИ** будет получено восьмеричное значение 0573, как показано ниже.

```
w1 ... 100 011 001    (0431)
w2 ... 001 101 010 | (0152)
-----
... 101 111 011    (0573)
```

Как и для поразрядного оператора **И**, не путайте операцию поразрядного **ИЛИ** (`|`) с логической операцией **ИЛИ** (`||`), которая используется для выполнения операция над двумя логическими значениями `true` и `false`.

Операцию поразрядного неисключающего **ИЛИ** часто называют просто поразрядное **ИЛИ** и обычно использует для установки заданных битов слова в единицу. Например, утверждение

```
w1 = w1 | 07;
```


установит три младших бита переменной `w1` в 1, независимо от состояния этих битов до начала операции. Разумеется, можно использовать и составной оператор присваивания, как показано ниже.

```
w1 |= 07;
```

Примеры по использованию операции неисключающее **ИЛИ** будут приведены далее в этой главе.

Поразрядный оператор `^`

Поразрядный оператор исключающее **ИЛИ** (`^`) работает следующим образом. Сравниваются соответствующие биты двух операндов, и если только один из битов равен 1, результат будет равен 1. А при равенстве обоих соответствующих битов или 0, или 1 результат будет равен 0, как показано в таблице истинности.

b1	b2	b1 ^ b2
0	0	0
0	1	1
1	0	1
1	1	0

Если для операндов `w1` и `w2` установлены восьмеричные значения 0536 и 0266 соответственно, то в результате применения операции исключающее **ИЛИ** будет получено восьмеричное значение 0750, как показано ниже.

```
w1 ... 101 011 110   (0536)
w2 ... 010 110 110 ^ (0266)
-----
... 111 101 000   (0750)
```

У этого оператора есть одно интересное свойство: если операцию исключающее **ИЛИ** использовать для одного и того же значения, то в результате будет получено нулевое значение. Этот прием часто использовался программистами на языке ассемблера как наиболее быстрый путь установить значение в нуль или сравнить два значения на их равенство. Этот способ не рекомендуется для использования в языке C, т.к. при этом скорость работы не повышается, а программа становится менее понятной.

Еще одно интересное применение оператора исключающее **ИЛИ** заключается в использовании его для перестановки значений двух переменных без выделения дополнительной памяти. Хорошо известно, что переставить значения двух переменных `i1` и `i2` можно с помощью следующей последовательности утверждений.

```
temp = i1;
i1 = i2;
i2 = temp;
```

Используя оператор исключающее **ИЛИ**, можно переставить значения переменных без использования дополнительной переменной `temp`.

```
i1 ^= i2;
i2 ^= i1;
i1 ^= i2;
```

В качестве упражнения вы можете проверить, что предыдущие утверждения корректно производят перестановку значений переменных `i1` и `i2`.

Поразрядный оператор ~

Оператор поразрядной инверсии является унарным оператором, и его действие заключается в инверсии всех битов операнда. Каждый бит операнда, который имел значение 1, становится 0, а значение 0 становится 1. Таблица истинности является очень простой и приводится только для полноты описания.

b1	~b1
0	1
1	0

Если переменная `w1` имеет тип `short int`, который занимает 16 битов, и ее значение равно 0122457, то применяя оператор поразрядной инверсии, получим в результате 0055320.

```
w1 1 010 010 100 101 111 (0122457)
w1 0 101 101 011 010 000 (0055320)
```

Не путайте оператор поразрядной инверсии (`~`) с арифметическим оператором унарного минуса (`-`) или с логическим оператором отрицания (`!`). Если переменная `w1` типа `int` имеет значение 0, то поставив перед ней минус, получим тот же 0. Применив к переменной `w1` операцию поразрядной инверсии, получим для всех битов значение 1, что можно трактовать как `-1`, т.е. знаковый бит будет иметь значение 1, а используя с переменной `w1` символ логического отрицания, получим 1, т.к. нулевое значение рассматривается как “ложь”, а любое другое значение рассматривается как “истина” (обычно это 1).

Оператор поразрядной инверсии особенно полезен в том случае, когда точно неизвестен размер типа операндов. Это позволит сделать программу более переносимой, или другими словами, менее зависимой от особенностей отдельного компьютера и поэтому корректно работающей на различных типах компьютеров. Например, для того чтобы установить младший бит переменной `w1` типа `int` в 0, можно использовать операцию поразрядного **И** для переменной `w1` и целого числа, состоящего из единиц, кроме младшего бита, который равен нулю. Поэтому следующее утверждение

```
w1 &= 0xFFFFF0;
```

корректно выполнит свою задачу только на тех компьютерах, у которых тип `int` составляет 32 бита. Но если заменить предыдущее утверждение следующим

```
w1 &= ~1;
```

то задача будет правильно выполнена на любых компьютерах, поскольку при расчете поразрядной инверсии для 1 будет учтено столько битов, сколько необходимо для заполнения всех битов, выделенных для данного типа.

В программе из листинга 12.2 суммируются все возможности бинарных операторов, представленных выше. Но сначала необходимо получить представление о старшинстве, или приоритетом, различных операторов.

Операторы поразрядного **И**, поразрядного **ИЛИ** и поразрядного неисключающего **ИЛИ** имеют более низкий приоритет, чем любые арифметические операторы, или операторы отношения, но более высокий приоритет, чем логические операторы **И** и **ИЛИ**. Бинарный оператор поразрядного **И** имеет приоритет выше, чем оператор поразрядного исключающего **ИЛИ**, который в свою очередь старше, чем оператор

поразрядного **ИЛИ**. Унарный оператор поразрядной инверсии имеет более высокий приоритет, чем любой бинарный оператор. Более подробно о старшинстве операторов рассказывается в приложении А, “Справочник по языку С”.

Листинг 12.2. Иллюстрация работы поразрядных операторов

```
/* Программа демонстрирует работу поразрядных операторов. */
#include <stdio.h>
int main (void)
{
    unsigned int w1 = 0525u, w2 = 0707u, w3 = 0122u;

    printf ("%o %o %o\n", w1 & w2, w1 | w2, w1 ^ w2);
    printf ("%o %o %o\n", ~w1, ~w2, ~w3);
    printf ("%o %o %o\n", w1 ^ w1, w1 & ~w2, w1 | w2 | w3);
    printf ("%o %o\n", w1 | w2 & w3, w1 | w2 & ~w3);
    printf ("%o %o\n", ~(~w1 & ~w2), ~(~w1 | ~w2));
    w1 ^= w2;
    w2 ^= w1;
    w1 ^= w2;
    printf ("w1 = %o, w2 = %o\n", w1, w2);

    return 0;
}
```

Листинг 12.2. Вывод

```
505 727 222
37777777252 37777777070 37777777655
0 20 727
527 725
727 505
w1 = 707, w2 = 525
```

Здесь необходимо выполнять каждую операцию с карандашом и бумагой для того, чтобы убедиться, что вы все хорошо понимаете и полученные вами результаты совпадают с теми, что приведены в книге. Программа должна запускаться на компьютерах, в которых для типа `int` выделяются 32 бита.

В четвертом утверждении `printf` важно учесть то, что бинарный оператор поразрядного **И** имеет более высокий приоритет, чем оператор поразрядного **ИЛИ**. В пятом утверждении `printf` демонстрируются правила де Моргана, а именно: выражение $\sim(a \& b)$ равно выражению $a | b$ и выражение $\sim(\sim a | \sim b)$ равно $a \& b$. Последовательность утверждений, которые следуют далее, приведена для проверки перестановки значений, что обсуждалось в разделе “Поразрядный оператор $\wedge$ ”.

Оператор `<<`

Когда оператор сдвига влево (`<<`) выполняется над некоторым значением, все биты, составляющие это значение, сдвигаются влево. Связанное с этим оператором число показывает количество битов, на которое значение должно переместиться. Биты, которые сдвигаются со старшего разряда, считаются потерянными, а на место младших битов всегда помещаются нули. Следовательно, если переменная `w1` имеет значение 3, то после выполнения утверждения

```
w1 = w1 << 1;
```

которое также можно написать след. образом

```
w1 <= 1;
```

в переменной `w1` будет находиться значение 6, которое получается после сдвига всех битов на одну позицию влево.

```
w1      ... 000 011 (03)
```

```
w1<<1   ... 000 110 (06)
```

Операнд слева для оператора “<<” является значением, которое должно быть сдвинуто, а значение справа показывает количество битов, на которое должен быть произведен сдвиг. Если вы сдвинете влево значение переменной `w1` еще раз, то получите восьмеричное значение 014, которое и будет являться новым значение переменной `w1`.

```
w1      ... 000 110 (06)
```

```
w1 << 1 ... 001 100 (014)
```

В действительности сдвиг влево на один разряд имеет эффект умножения числа на два. На самом деле компилятор языка C выполняет умножение числа на степени числа два с помощью сдвига числа на соответствующее число позиций (значение степени двойки), поскольку операция сдвига выполняется значительно быстрее, чем операция перемножения чисел на всех компьютерах.

Листинг с программой, демонстрирующей операции сдвига влево, представлен после описания оператора сдвига вправо.

Оператор >>

Как можно понять из названия оператора сдвига вправо (>>), это оператор используется для операций сдвига вправо значения переменной. Биты, которые сдвигаются с самого младшего бита, теряются. Заполнение освободившихся старших битов зависит от типа значения, которое сохраняется в переменной. Если это беззнаковое целое число, то старшие биты заполняются нулями. Если же производится сдвиг вправо числа со знаком, то заполнение старших битов зависит от знака, которое имеет значение, и от того, как эта операция реализована на данном компьютере.

Если знаковый бит имеет значение 0 (положительное число), то этот нуль и сдвигается вправо, независимо от компьютера. Но если знаковый бит имеет значение 1 (отрицательное число), то на некоторых компьютерах эта единица будет сдвигаться, а на других — нет, и заполнение будет производиться нулем. Первый тип этой операции называется *арифметическим сдвигом вправо*, а второй тип называется *логическим сдвигом вправо*.

Никогда не рассчитывайте на то, что система будет иметь определенный тип сдвига вправо, арифметический или логический. Программа, которая производит сдвиг вправо, должна корректно работать на любом типе компьютеров, а не выполняться на одном компьютере и давать сбой на другом.

Предположим, что переменная `w1` имеет тип `unsigned int`, который представлен 32 битами, и для переменной `w1` задано шестнадцатеричное значение `F777EE22`, которое сдвигается вправо на одну позицию с помощью утверждения

```
w1 >>= 1;
```

что приводит к тому, что переменная `w1` будет иметь шестнадцатеричное значение `7BBBF711`.

```
w1      1111 0111 0111 0111 1110 1110 0010 0010 (F777EE22)
w1>>1   0111 1011 1011 1011 1111 0111 0001 0001 (7BVBVF711)
```

Если же переменная объявлена как тип `int` (знаковая), то та же самая операция будет выполнена по-разному в зависимости от типа компьютера и на некоторых компьютерах в результате может получиться значение `FVBVF711`, т.е. операция будет выполнена как арифметический сдвиг.

Обратите внимание, что в языке программирования `C` не определяется результат попытки сдвига вправо или влево на количество позиций, превышающих или равных количеству битов для объявленного типа переменной. Поэтому, если на компьютере целое число представлено 32 битами, а сдвиг производится на 32 или более позиций, то ничего нельзя сказать о результате. Также отметим еще раз, что если сдвигается отрицательное значение, то результат также будет неопределенным.

Функция сдвига

Теперь мы рассмотрим работу операторов сдвига влево и вправо в реальной программе, для чего обратимся к программе из листинга 12.3. Некоторые компьютеры выполняют команды сдвига влево и вправо в зависимости от значения счетчика, которое может быть положительным или отрицательным. Поэтому напомним программу, реализующую функцию сдвига подобно этим командам, т.е. если значение счетчика положительное, то сдвиг производится влево, а если значение счетчика отрицательное, то сдвиг производится вправо. Количество позиций сдвига зависит от абсолютного значения счетчика.

Листинг 12.3. Реализация функций сдвига

```
// Функция сдвигает беззнаковое целое значение влево
// если счетчик имеет положительное значение, и вправо,
// при отрицательном значении счетчика.
#include <stdio.h>
unsigned int shift (unsigned int value, int n)
{
    if ( n > 0 )           // Сдвиг влево.
        value <<= n;
    else                   // Сдвиг вправо.
        value >>= -n;

    return value;
}

int main (void)
{
    unsigned int w1 = 0177777u, w2 = 0444u;

    unsigned int shift (unsigned int value, int n);
    printf ("%o\t%o\n", shift (w1, 5), w1 << 5);
    printf ("%o\t%o\n", shift (w1, -6), w1 >> 6);
    printf ("%o\t%o\n", shift (w2, 0), w2 >> 0);
    printf ("%o\n", shift (shift (w1, -3), 3));

    return 0;
}
```

Листинг 12.3. Вывод

```
7777740 7777740
1777 1777
444 444
177770
```

В функции `shift`, описанной в листинге 12.3, объявлен аргумент `value` типа `unsigned int`, при этом гарантируется, что при правом сдвиге значения переменной `value` заполнение старших битов будет производиться нулями, другими словами, будет выполняться логический сдвиг вправо.

Если значение аргумента `n`, которое является счетчиком позиций, больше нуля, то функция производит сдвиг влево на `n` позиций для значения переменной `value`. Если значение `n` отрицательное или равно нулю, функция выполняет сдвиг вправо, где количество позиций сдвига определяется абсолютным значением `n`. При первом вызове функции `shift` в функции `main` производится сдвиг значения переменной `w1` влево на пять битов. Функция `printf` вызывается для отображения результатов вызова функции `shift` и результата непосредственного сдвига переменной `w1` влево на пять битов влево для того, чтобы можно было сравнить эти значения.

При втором вызове функции `shift` производится сдвиг переменной `w1`. Результат, полученный при вызове функции `shift`, идентичен результату, полученному при непосредственном сдвиге переменной на шесть позиций вправо, что видно из вывода, сделанного программой.

В третьем случае при вызове функции `shift` должен быть произведен сдвиг на нуль позиций. При этом функция `shift` производит правый сдвиг на нуль позиций, при котором исходное значение переменной не изменяется, что и видно из вывода, сделанного программой.

Последний вызов функции `printf` демонстрирует вложение вызовов функций `shift`. Внутренняя функция `shift` выполняется первой. Это приводит к тому, что переменная `w1` будет сдвинута вправо три раза. Результатом вызова этой функции будет значение `017777`, которое передается во внешнюю функцию `shift` и сдвигается влево на три позиции. Как можно видеть из вывода, сделанного программой, это имеет эффект установки трех младших битов переменной `w1` в значение `0`. Разумеется, это же самое можно сделать проще с помощью операции поразрядного И переменной `w1` с инверсным значением для `7 (~7)`.

Ротация битов

В следующем примере объединены вместе некоторые операции с битами для разработки функции ротации значения влево или вправо. Процесс ротации подобен сдвигу за исключением того, что биты, которые сдвигаются с самых крайних битов, не теряются, а переносятся на противоположную сторону. То есть значение старшего бита при сдвиге влево переносится в самый младший бит после того, как значение младшего бита сдвигается влево. Соответственно, при сдвиге вправо, значение самого младшего бита переносится в самый старший бит.

Следовательно, если вы работаете на 32-разрядной машине, то значение шестнадцатеричного числа `80000000`, для которого была сделана ротация влево на одну позицию, приведет в результирующему значению `00000001`, поскольку единичное значение самого старшего бита не будет потеряно, а переместится на место самого младшего бита. В этом и заключается смысл ротации.

Функция ротации принимает два аргумента: первый — это значение, которое должно быть подвержено ротации, и второе — число битов, на которое должна быть сделана ротация. Если второй аргумент положительный, то ротация будет произведена влево, в противном случае, ротация будет выполняться вправо. Для реализации этой функции можно использовать довольно прямолинейный подход. Например, для того чтобы выполнить ротацию переменной *x* влево на *n* битов, когда переменная *x* представляет тип *int*, а переменная *n* может принимать значение от нуля до числа битов, выделенных для переменной *x* минус единица, вы можете извлечь старшие *n* битов, произвести сдвиг влево и затем вставить эти *n* бит на место освободившихся младших битов. Аналогично можно действовать и при сдвиге вправо.

В программе из листинга 12.4 реализована функция с именем *rotate*, в которой реализован описанный выше алгоритм. В этой функции сделано предположение, что тип *int* занимает 32 бита.

В упражнении в конце главы показан способ написания подобных функций, который позволяет исключить эти ограничения.

Листинг 12.4. Реализация функции ротации переменной

```
// Программа для иллюстрации ротации целочисленных переменных
#include <stdio.h>
int main (void)
{
    unsigned int w1 = 0xabcdef00u, w2 = 0xffff1122u;
    unsigned int rotate (unsigned int value, int n);

    printf ("%x\n", rotate (w1, 8));
    printf ("%x\n", rotate (w1, -16));
    printf ("%x\n", rotate (w2, 4));
    printf ("%x\n", rotate (w2, -2));
    printf ("%x\n", rotate (w1, 0));
    printf ("%x\n", rotate (w1, 44));

    return 0;
}

// Функция для ротации целочисленного значения типа
// unsigned int влево или вправо.
unsigned int rotate (unsigned int value, int n)
{
    unsigned int result, bits;

    // Ограничение диапазона ротации.
    if ( n > 0 )
        n = n % 32;
    else
        n = -(-n % 32);
    if ( n == 0 )
        result = value;
    else
        if ( n > 0 ) {          // Ротация влево.
            bits = value >> (32 - n);
            result = value << n | bits;
        }
}
```

```

    else {                                // Ротация вправо.
        n = -n;
        bits = value << (32 - n);
        result = value >> n | bits;
    }
    return result;
}

```

Листинг 12.4. Вывод

```

cdef00ab
ef00abcd
fff1122f
bfffc448
abcdef00
def00abc

```

В функции производится проверка того, что значение переменной *n* находится в допустимом диапазоне, для чего написан следующий код

```

if ( n > 0 )
    n = n % 32;
else
    n = -(-n % 32);

```

где сначала производится проверка положительного значения *n*, при этом вычисляется значение *n* переменной по модулю, представляющему число битов, составляющих тип *int* (в нашем случае это 32 бита). Рассчитанное значение присваивается обратно переменной *n*. Это гарантирует, что значение *n* будет находиться в диапазоне от 0 до 31. Если переменная *n* имеет отрицательное значение, то оно сначала приводится к положительному значению, поскольку в языке программирования C четко не определена операция деления по модулю для отрицательных чисел и на различных машинах это может быть или положительное, или отрицательное значение. После выполнения деления по модулю число опять приводится к отрицательному значению и будет находиться в диапазоне от -31 до 0. Если значение для количества позиций сдвига равно 0, то в функции просто присваивается значение *value* результирующему значению. В противном случае производится ротация значения.

Ротация переменной на *n* битов влево производится в соответствии с алгоритмом, состоящим из трех шагов. Сначала извлекаются левые *n* битов исходного значения в отдельную переменную, которая сдвигается вправо на число позиций (32-*n*). Затем исходное значение сдвигается влево на *n* битов и наконец, над получившимся значением и извлеченными битами производится операция поразрядного И (&). Аналогичная процедура выполняется и при сдвиге вправо.

Обратите внимание, что в процедуре *main* в этот раз используется шестнадцатеричная нотация. Это не принципиально и для разнообразия программ можно также использовать и восьмеричные числа. При первом вызове функции *rotate* определено, что значение переменной *w1* должно быть сдвинуто на восемь битов влево. Как можно видеть из вывода, сделанного программой, функцией *rotate* возвращается значение *cdef00ab*, которое является значением *abcdef00*, сдвинутым влево на восемь позиций.

Во втором случае вызова функции *rotate* производится ротация значения переменной *w1* на 16 битов вправо. Следующие два вызова функции *rotate* производят

аналогичные вещи и не требуют объяснения. Последний перед концом вызов функции `rotate` использует сдвиг на 0 позиций. При этом из вывода, сделанного программой, видно, что просто возвращается исходное значение.

При последнем вызове функции `rotate` задается сдвиг влево на 44 позиции. Это имеет эффект сдвига влево на 12 битов ($44 \% 32$ будет 12).

Битовые поля

Используя все битовые операторы, о которых говорилось выше, можно выполнять довольно сложные преобразования с битами. Битовые операции часто выполняются над элементами, которые содержат упакованную информацию. Так же, как тип `short int` может использоваться для хранения данных в памяти компьютера, для хранения данных можно использовать и биты одного байта или слова, при этом не надо расходовать дополнительные биты, составляющие байт или слово. Например, при булевых операциях используются только два значения, 0 или 1, которые можно использовать как флаги. Объявляя переменную типа `char` (8 битов) для хранения одного флага, что делается на большинстве компьютеров, мы будем расходовать впустую 7 битов на каждом флаге. Часто необходимо хранить большое количество флагов в таблице и при этом получается большой расход памяти, если для каждого флага использовать тип `char`.

Для упаковки информации в целях более экономного расходования памяти в языке программирования C можно использовать два метода. Первый — это использовать для хранения нескольких данных простой тип `int`, а доступ к отдельным битам производить с помощью операций над битами, о которых говорилось выше. Второй — это задать структуру для упакованной информации, которую в языке программирования C называют *битовым полем*.

Для демонстрации того, как можно использовать первый метод, запакуют пять значений данных в слово, поскольку надо будет создавать большую таблицу данных в памяти. Предположим, что три значения из данных являются флагами, которые обозначим `f1`, `f2` и `f3`; четвертое значение является целым числом в диапазоне от 1 до 255. Его назовем `type` и последнее значение назовем `index`, диапазон изменений которого будет простираться от 0 до 100 000.

Хранение значений флагов `f1`, `f2` и `f3` требует только трех битов памяти, по одному биту для значений `true/false` в каждом флаге. Хранение значения для переменной `type` требует восьми битов, т.к. для диапазона от 1 до 255 требуется 1 байт памяти. Наконец, сохранение целочисленной переменной `index`, которая может принимать любое значение в диапазоне от 0 до 100 000, требует 18 битов. Поэтому общий размер занимаемой памяти для пяти переменных `f1`, `f2`, `f3`, `type` и `index` требует 29 битов. Исходя из этого можно задать одну целочисленную переменную, которая будет содержать все пять значений, следующим образом

```
unsigned int packed_data;
```

и затем произвольно распределить последовательности (поля) битов для хранения пяти значений данных в переменной `packed_data`. Одно из возможных распределений показано на рис. 12.1, где предполагается, что размер переменной `packed_data` составляет 32 бита.

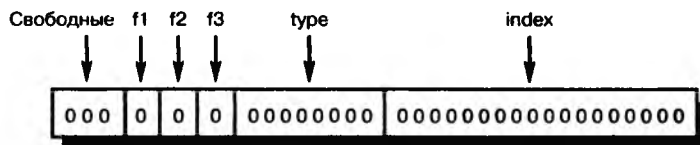


Рис. 12.1. Распределение битов в переменной `packed_data`.

Обратите внимание, что при этом остаются неиспользованными три бита. Сейчас можно разработать корректные последовательности битовых операций для переменной `packed_data` для присваивания и извлечения значений из различных полей переменной. Например, можно присвоить полю `type` значение 7 с помощью сдвига значения 7 на соответствующее число позиций влево и затем произвести операцию поразрядного **ИЛИ** для этого значения и переменной `packed_data`.

```
packed_data |= 7 << 18;
```

Или можно присвоить полю `type` значение `n`, где `n` находится в диапазоне от 0 до 255, с помощью следующего утверждения.

```
packed_data |= n << 18;
```

Для того чтобы убедиться, что значение `n` находится в заданном диапазоне, можно выполнить операцию поразрядного **И** для `n` и шестнадцатеричного значения `0xff`.

Разумеется, предыдущее утверждение работает только в том случае, если известно, что поле `type` имеет значение 0. Другими словами, необходимо установить для поля `type` значение 0 перед тем, как присваивать ему другое значение. Это можно сделать с помощью операции поразрядного **И** для переменной `packed_data` и константы, имеющей нулевое значение для этого поля, и установленных в единицу остальных битов, как показано ниже. Такая константа часто называется *маской*.

```
packed_data &= 0xfc03ffff;
```

Для того чтобы явно не рассчитывать константу и избежать при этом возможных ошибок, а также сделать эту операцию независимой от размера типа, вместо предыдущего утверждения можно использовать следующее утверждение, устанавливающее поле `type` в нуль.

```
packed_data &= ~(0xff << 18);
```

Комбинируя описанные выше утверждения, вы можете установить для поля `type` переменной `packed_data` значение, занимающее не более 8 битов из младших байтов, независимо от предыдущего состояния поля, с помощью следующего утверждения.

```
packed_data = (packed_data & ~(0xff << 18)) | ((n & 0xff) << 18);
```

В этом утверждении некоторые круглые скобки являются излишними и добавлены с целью более удобного восприятия.

Из предыдущего выражения видно, что с помощью относительно простого выражения можно выполнить такую задачу, как присваивание полю `type` заданного значения. Извлечение значения из того поля также не будет очень сложным. Сначала поле должно быть сдвинуто влево и занять младшие биты, а затем должна быть произведена операция поразрядного **И** с константой, представляющей соответствующую маску. Таким образом, извлечение значения из поля `type` для переменной `packed_data` и присваивание его переменной `n` выполняется с помощью следующего утверждения.

```
n = (packed_data >> 18) & 0xff;
```

Но в языке программирования С существует и более удобный способ для работы с битовыми полями. При этом используется специальный синтаксис при описании структуры, который позволяет задавать битовые поля и присваивать им имена. Когда в языке программирования С используется термин “битовые поля”, он имеет отношение именно к этому случаю.

Для создания битового поля, о котором говорилось выше, можно, например, описать структуру с именем `packed_struct`, как это сделано ниже.

```
struct packed_struct
{
    unsigned int :3;
    unsigned int f1:1;
    unsigned int f2:1;
    unsigned int f3:1;
    unsigned int type:8;
    unsigned int index:18;
};
```

Структура `packed_struct` описана как содержащая шесть членов. Первый член не имеет имени. Символ “:3” задает три безымянных бита. Второй член, `f1`, также имеет тип `unsigned int`. Символы “:1”, которые непосредственно следуют за именем, говорят о том, что в данном члене будет храниться 1 бит. Аналогично, объявленные флаги `f2` и `f3` также рассчитаны на хранение по одному биту. Член с именем `type` занимает 8 битов, тогда как член `index` рассчитан на хранение 18 бит.

Компилятор языка программирования С автоматически объединяет все эти поля. Приятным в данном случае является то, что к полям, описанным для структуры `packed_struct`, обращаться так же удобно, как и к членам обычной структуры. То есть если объявить переменную типа `packed_data` следующим образом.

```
struct packed_struct packed_data;
```

то очень просто присвоить значение 7 полю `type` переменной `packed_data` с помощью следующего утверждения.

```
packed_data.type = 7;
```

разумеется, можно присвоить этому полю и значение переменной `n` с помощью аналогичного утверждения.

```
packed_data.type = n;
```

Причем в этом случае нет необходимости беспокоиться о том, что значение переменной `n` будет слишком большим. Только младшие 8 битов будут учитываться при присваивании значения для поля `packed_data.type`.

Извлечение значения из поля также выполняется без особых усилий со стороны программиста, для чего можно использовать простое утверждение

```
n = packed_data.type;
```

которое извлечет значение поля `type` переменной `packed_data` и присвоит его переменной `n`, при этом автоматически будет произведен сдвиг в сторону младших бит.

Битовые поля можно без проблем использовать в обычных выражениях, они будут автоматически преобразованы в необходимый тип. Поэтому утверждение

```
i = packed_data.index / 5 + 1;
```

является полностью корректным, как и утверждение

```
if ( packed_data.f2 )
```

```
...
```

которое проверяет флаг поля f2 на истину или ложь. Но одна проблема все же существует, т.к. нет стандартного порядка распределения полей: слева направо или справа налево. Это не представляет проблемы до тех пор, пока вы не начнете работать с данными, созданными другой программой или на другой машине. В этом случае вы должны знать, как выполнено распределение полей. Возможно, придется описать структуру `packed_struct` следующим образом.

```
struct packed_struct
{
    unsigned int index:18;
    unsigned int type:8;
    unsigned int f3:1;
    unsigned int f2:1;
    unsigned int f1:1;
    unsigned int :3;
};
```

Это необходимо для того, чтобы получить распределение полей, как показано на рис. 12.1. Никогда не делайте предположений о распределении полей структур в памяти о том, включают ли они битовые поля или нет.

Также можно включать стандартные типы данных в структуры, которые содержат битовые поля. При желании можно описать структуру, которая содержит типы `int` и `char`, а также два однобитовых флага. Следующее описание структуры полностью корректно.

```
struct Листинг_entry
{
    int count;
    char c;
    unsigned int f1:1;
    unsigned int f2:1;
};
```

Необходимо сделать еще несколько замечаний относительно битовых полей. Битовые поля могут быть объявлены только как тип `integer` или `_Bool`. Если битовое поле имеет тип `int`, то этот знаковый или беззнаковый тип зависит от реализации. Для исключения неоднозначности используйте явные объявления: `signed int` или `unsigned int`. Битовые поля нельзя объединять в массивы, поэтому нельзя иметь массив битовых полей, такой как `flag:1[5]`. Наконец, нельзя использовать адрес битового поля, и поэтому не может быть такого типа, как "указатель на битовое поле".

Битовые поля упакованы в блоки в порядке, в котором они описаны при объявлении структуры, при этом размер блока определяется реализацией и, вероятнее всего, совпадает с размером слова. Компилятор языка программирования C не переупорядочивает битовые поля для получения более оптимального распределения памяти.

Но в некоторых случаях может производиться выравнивание за счет безымянного поля. Это может использоваться для выравнивания следующего поля структуры по границе блока.

На этом заканчивается обсуждение операций с битами в языке программирования C. Теперь вы можете представить всю мощь и гибкость языка C для эффективного выполнения операций с битами. Операторы удобны и понятны при выполнении поразрядных операций **И**, неисключающем и исключающем **ИЛИ**, инверсии, левом и правом сдвигах. Удобный синтаксис для работы с битовыми полями позволяет выделять заданное количество битов для данных, которые легко задавать и извлекать без использования операций маскирования и сдвигов.

Обратитесь к главе 14, “Еще о типах данных”, для знакомства с отдельными ситуациями при выполнении поразрядных операций с двумя различными целочисленными типами данных, например, между `unsigned long int` и `short int`.

Упражнения

1. Наберите и выполните четыре программы, представленные в этой главе. Сравните выводы, сделанные каждой программой с выводами, приведенными после каждой программы в тексте.
2. Напишите программу, которая определяет, как отдельный компьютер выполняет сдвиг вправо: арифметически или логически.
3. Используя выражение `~0`, которое создает целое число из одних единиц, напишите функцию с именем `int_size`, которая возвращает кол-во битов, содержащихся в типе `int` для отдельной машины.
4. Используя результат, полученный в упражнении 3, модифицируйте функцию `rotate` из листинга 12.4 таким образом, чтобы не делать предположений относительно размера типа `int`.
5. Напишите функцию с именем `bit_test`, которая принимает два аргумента: число типа `unsigned int` и номер бита `n`. Используйте функцию для получения состояния бита с номером `n`. При этом сделайте так, чтобы номер 0 ссылался на самый младший бит. Дополнительно напишите функцию с именем `bit_set`, которая также принимает два аргумента: число типа `unsigned int` и номер бита `n`. Используйте эту функцию для изменения состояния бита с номером `n`.
6. Напишите функцию с именем `bitpat_search`, которая ищет заданный шаблон внутри переменной типа `unsigned int`. Функция должна принимать три аргумента и ее вызов должен производиться следующим образом.

```
bitpat_search (source, pattern, n)
```

При этом функция производит поиск заданного шаблона, это правые `n` битов аргумента `pattern`, в целочисленной переменной `source`. Если совпадение обнаружено, функция должна возвращать номер бита, с которого начинается совпадение. При этом самый младший бит имеет номер 0. Если совпадения не найдено, то возвращается -1. Например, при вызове

```
index = bitpat_search (0xelf4, 0x5, 3);
```

будет произведен поиск в числе `0xelf4` (двоичное `1110 0001 1111 0100`) на предмет наличия последовательности `0x5` (двоичное `101`). Функция возвратит число 11, которое свидетельствует о том, что совпадение найдено и начинается с бита номер 11.

Постарайтесь сделать так, чтобы не надо было делать никаких предположений о размере типа `int` (см. упр. 3).

7. Напишите функцию с именем `bitpat_get`, которая извлекает заданную последовательность битов из заданного числа. Функция должна принимать три аргумента: первый — это число типа `unsigned int`, второй аргумент типа `int` задает начальный бит, а третий аргумент указывает кол-во битов. Используйте правило, согласно которому самый младший бит имеет номер 0. Извлечение последовательности битов производится из первого аргумента и результат извлечения возвращается. Поэтому при следующем вызове

```
bitpat_get (x, 0, 3)
```

будут извлечены три младших бита из переменной `x`. А при вызове

```
bitpat_get (x, 3, 5)
```

будут извлечены пять битов начиная с четвертого бита слева.

8. Напишите функцию с именем `bitpat_set` для установки в заданные биты соответствующего значения. Функция должна принимать четыре аргумента: указатель на тип `unsigned int`, в который должно быть занесено заданное значение, аргумент типа `unsigned int`, содержащий значение, которое должно быть установлено и выровнено по правой стороне, значение типа `int`, определяющее начальный бит (крайний левый бит имеет значение 0), и наконец, значение типа `int`, задающее размер поля. Поэтому следующий вызов

```
bitpat_set (&x, 0, 2, 5);
```

должен установить 5 битов в переменной `x`, начиная с третьего бита слева (бит с номером 2), в нуль. А вызов

```
bitpat_set (&x, 0x55u, 0, 8);
```

должен установить для восьми левых битов переменной `x` шестнадцатеричное значение 55.

При этом не должно делаться никаких предположений о размере типа `int` (см. упр. 3).

13

Препроцессор

В этой главе описываются еще несколько уникальных особенностей языка программирования C. Препроцессор языка C предлагает дополнительные возможности, которые позволяют программисту быстрее и легче разрабатывать программы, легче их читать, модифицировать и переносить программу на другие системы. Препроцессор можно использовать для настройки языка программирования под отдельное приложение или стиль программиста.

Препроцессор является частью процесса компиляции, при котором распознаются специальные утверждения, представленные в программе на языке C. При обнаружении таких утверждений препроцессор производит их анализ и обработку, прежде чем будет проведена компиляция программы. Утверждения для препроцессора распознаются по специальному символу “#”, который должен располагаться в строке перед утверждением. Как можно убедиться, утверждения для препроцессора почти не отличаются от обычных утверждений языка программирования C. Начнем изучение препроцессора с утверждения `#define`.

Утверждение `#define`

Одним из основных применений утверждения `#define` является присваивание символических имен константам программы. Следующее утверждение

```
#define YES 1
```

введет имя `YES` и сделает его равным 1. Имя `YES` может затем использоваться в программе везде, где необходимо использовать константу 1.

Когда в программе встречается это имя, препроцессор автоматически подставляет в программу вместо него значение 1. Например, в программе на языке C может встретиться следующее утверждение, в котором используется символическое имя `YES`.

```
gameOver = YES;
```

В этом утверждении присваивается значение `YES` переменной `gameOver`. При этом нет необходимости акцентировать внимание на том, какое значение скрывается под именем `YES`, поскольку смысл утверждения полностью понятен. Но поскольку вы знаете, что под символическим именем `YES` подразумевается значение 1, то именно это значение и будет присвоено переменной `gameOver`. Препроцессорное утверждение

```
#define NO 0
```


задает имя NO, которое в программе будет заменено на заданное значение 0. Поэтому в утверждении

```
gameOver = NO;
```

происходит присваивание значения для символического имени NO переменной gameOver, а в утверждении:

```
if ( gameOver == NO )
```

```
...
```

сравнивается значение переменной gameOver со значением символического имени NO. Только в одном случае вы не сможете использовать замену символического имени — внутри символьной строки. Поэтому утверждение

```
char *charPtr = "YES";
```

установит для переменной charPtr значение "YES", а не значение "1".

Символическое имя не является переменной, поэтому ему нельзя присвоить значение, за исключением момента, когда это имя задается и значение используется в утверждении после ключевого слова #define, после чего это имя используется в программе и обрабатывается препроцессором. Это аналогично процессу поиска и замены в текстовом редакторе, когда все вхождения некоторой последовательности символов заменяются другой последовательностью.

Обратите внимание, что утверждение с ключевым словом #define имеет специальный синтаксис. Здесь нет знака равенства, используемого в других выражениях присваивания. Кроме того, точка с запятой не ставится в конце утверждения. Скоро вы поймете, почему используется такой синтаксис. Но сначала рассмотрим небольшую программу с использованием ранее описанных имен YES и NO. Функция isEven из листинга 13.1 просто возвращает YES, если аргумент четный, и NO, если аргумент нечетный.

Листинг 13.1. Использование утверждения #define

```
#include <stdio.h>
#define YES 1
#define NO 0
// Функция определяет четность или нечетность
// целочисленного значения.
int isEven (int number)
{
    int answer;
    if ( number % 2 == 0 )
        answer = YES;
    else
        answer = NO;

    return answer;
}

int main (void)
{
    int isEven (int number);
    if ( isEven (17) == YES )
        printf ("yes ");
    else
```

```

    printf ("no ");
    if ( isEven (20) == YES )
        printf ("yes\n");
    else
        printf ("no\n");

    return 0;
}

```

Листинг 13.1. Вывод

```
no yes
```

Обычно первыми в программе располагаются утверждения `#define`, хотя это и не обязательно, т.е. они могут находиться в любом месте программы. Единственное правило, которое необходимо соблюдать, — это чтобы символическое имя было задано в программе прежде, чем оно будет использоваться. Поведение символического имени отличается от поведения переменных. Здесь нет такого понятия, как локальное символическое имя. После того как имя определено в программе, независимо от того, внутри или вне функции, оно будет учитываться в любом месте программы. Большинство программистов группируют объявления символических имен в начале программы (или внутри подключаемого файла), с тем чтобы к ним можно было легко обратиться и распространить на один или несколько исходных файлов. Следует всегда предварительно ознакомиться с тем, как использовать специальные заголовочные (подключаемые) файлы в программе.

Например, символическое имя `NULL` очень часто используется в программах и определяет нулевой указатель. Это имя уже объявлено в заголовочном файле с именем `stddef.h`.

Используя в программе уже рассмотренное утверждение

```
#define NULL 0
```

можно писать гораздо более понятные выражения, наподобие представленным ниже.

```
while ( listPtr != NULL )
...

```

Такое выражение задает условие выхода из цикла с помощью неравенства `listPtr!=NULL`, т.е. выход из цикла произойдет тогда, когда переменной `listPtr` не будет на что указывать (нулевой указатель).

Рассмотрим еще один пример использования символических имен. Предположим, вы хотите написать три функции для определения площади круга, длины окружности и поверхности сферы заданного радиуса. Поскольку во всех трех функциях используется константа, которую трудно запомнить, то имеет смысл задать эту константу в начале программы и затем использовать ее значения по мере необходимости в каждой функции. (Хотя можно и не задавать значение для `PI`, которое уже задано в заголовочном файле `math.h`. Подключая непосредственно это — файл к своей программе, вы можете обращаться к этому значению с помощью символического имени `M_PI`).

В программе из листинга 13.2 показано, как задавать константу, которую затем можно использовать в программе.

Листинг 13.2. Еще об утверждении #define

```
/* Функция для расчета площади круга, длины
   окружности и поверхности сферы заданного радиуса. */
#include <stdio.h>
#define PI 3.141592654
double area (double r)
{
    return PI * r * r;
}
double circumference (double r)
{
    return 2.0 * PI * r;
}
double volume (double r)
{
    return 4.0 / 3.0 * PI * r * r * r;
}

int main (void)
{
    double area (double r), circumference (double r),
           volume (double r);
    printf ("radius = 1: %.4f %.4f %.4f\n",
           area(1.0), circumference(1.0), volume(1.0));
    printf ("radius = 4.98: %.4f %.4f %.4f\n",
           area(4.98), circumference(4.98), volume(4.98));

    return 0;
}
```

Листинг 13.2. Вывод

```
radius = 1: 3.1416 6.2832 4.1888
radius = 4.98: 77.9128 31.2903 517.3403
```

В самом начале программы для константы PI задано значение 3.141592654. Использование символического имени при расчетах площади круга, длины окружности и площади поверхности приводит к автоматической замене символического имени тем значением, которое ему присвоено.

Несомненно, символическое имя запоминается значительно легче чем числовое значение. Также не менее важно, что при необходимости изменения значения числовой константы (если первое значение было ошибочным), которая была присвоена символическому имени, ее надо будет поменять только в одном месте программы, в утверждении #define, а не искать все вхождения этого значения и производить замены. К тому же при поиске можно пропустить некоторые значения или допустить ошибку при замене и, соответственно, получить неправильные результаты.

Возможно, вы уже обратили внимание на то, что все используемые символические имена (YES, NO, NULL и PI) написаны заглавными буквами. Это сделано специально, с тем чтобы они отличались от обычных переменных. Среди программистов принято негласное соглашение — писать все символические имена заглавными буквами, поскольку при этом легко определить, является ли имя символическим или это просто переменная.

Еще одним негласным правилом является использование префиксов, например, для символических имен используется буква `k`. В этом случае можно не использовать в имени все заглавные буквы, и выделять только первые буквы слова. Например, имена `kMaximumValues` и `kSignificantDigits` вполне соответствуют этому правилу.

Модифицируемые программы

Использование символических имен позволяет сделать программы более модифицируемыми. Например, когда вы задаете массив, вы должны определить количество элементов массива — явно или неявно (с помощью списка инициализации). В дальнейшем предварительное знание количество элементов массива будет использоваться в утверждениях программы. Например, если массив `dataValues` задан в программе следующим образом

```
float dataValues[1000];
```

то, вероятнее всего, вы будете использовать в программе утверждение, в котором необходимо указать число элементов массива `dataValues`. Например, в цикле `for`

```
for ( i = 0; i < 1000; ++i )  
    ...
```

необходимо использовать значение 1000 для задания верхней границы цикла, в котором просматриваются все элементы. А в утверждении `if`

```
if ( index > 999 )  
    ...
```

производится проверка выхода индекса за границы диапазона, т.е. проверяется превышение значения верхней границы массива.

А теперь предположим, что вы изменили размер массива `dataValues` с 1000 элементов до 2000. При этом вам придется изменить все утверждения, в которых используется это значение, т.е. изменить все значения, которые соответствуют размеру массива `dataValues` в 1000 элементов, и не изменять другие значения, которые также могут быть равны 1000. Как видим, задача весьма не простая.

Гораздо целесообразнее будет задать символическое имя для верхней границы массива, что сделает программу более легкой для понимания и модификации. Если при этом в утверждении `#define` вы зададите понятное символическое имя, например, `MAXIMUM_DATAVALUES`:

```
#define MAXIMUM_DATAVALUES 1000
```

то вы можете использовать это имя и при объявлении массива `dataValues`, как показано в следующем утверждении.

```
float dataValues[MAXIMUM_DATAVALUES];
```

Во всех последующих утверждениях вы тоже можете использовать это символическое имя. Например, в утверждении `for` можно задать верхнюю границу диапазона массива `dataValues`.

```
for ( i = 0; i < MAXIMUM_DATAVALUES; ++i )  
    ...
```

А для проверки выхода индекса за границы диапазона можно написать следующее утверждение.

```
if ( index > MAXIMUM_DATAVALUES - 1 )
```

```
...
```

Также немаловажной является возможность легко изменять размеры массива `dataValues`. Все изменения коснутся только одного утверждения и не надо будет проверять всю программу. Необходимо только поставить значение 2000 в утверждении `#define`.

```
#define MAXIMUM_DATAVALUES 2000
```

И если во всех остальных случаях в программе используется символическое имя `MAXIMUM_DATAVALUES`, а не непосредственное значение, то это изменение будет единственным. Именно таким образом можно сделать программы легко понимаемыми и модифицируемыми.

Переносимые программы

Еще одним полезным свойством символических имен является то, что они позволяют сделать программу переносимой с одной системы на другую. Как известно, при этом приходится использовать константы для настройки особенностей компьютера, на котором программа запускается. Например, для учета структуры памяти, идентификаторов или количества битов, составляющих слово. Вспомните функцию `rotate` из листинга 12.4, где учитывался тот факт, что тип `int` содержит 32 бита для того компьютера, на котором программа должна выполняться.

Если вы будете выполнять эту программу на другом компьютере, в котором тип `int` содержит 64 бита, то функция не будет работать правильно. Разумеется, вы можете написать функцию так, чтобы она сама определяла количество битов в типе `int`, и тогда она будет полностью независима от системы. Вернитесь к упражнениям 3 и 4 из главы 12, "Операции с битами". Изучите нижеприведенный код. В ситуации, когда программа должна использовать независимые от системы значения, имеет смысл изолировать такие значения от программы, если это возможно. И утверждение `#define` окажет в этом значительную помощь. В новой версии функции `rotate` можно не опасаться, что при запуске на другой системе она будет работать неправильно, поскольку введены несложные улучшения. Ниже приведена версия этой функции.

```
#include <stdio.h>
#define kIntSize 32 // *** Настройка на систему !!! ***
// Функция циклически сдвигает целое беззнаковое
// значение типа unsigned int влево или право
unsigned int rotate (unsigned int value, int n)
{
    unsigned int result, bits;
    /* определяем величину сдвига */
    if ( n > 0 )
        n = n % kIntSize;
    else
        n = -(-n % kIntSize);
    if ( n == 0 )
        result = value;
    else if ( n > 0 ) /* сдвиг влево */
    {
        bits = value >> (kIntSize - n);
```

```

    result = value << n | bits;
}
else          /* сдвиг вправо */
{
    n = -n;
    bits = value << (kIntSize - n) ;
    result = value >> n | bits;
}
return result;
}

```

Более развитые возможности

При задании символического имени можно использовать не только простые константы. Вполне возможно включать целые выражения и, как вы скоро увидите, кое-что еще!

В следующем выражении задается имя TWO_PI, за которым будет скрываться результат произведения 2.0 и 3.141592654:

```
#define TWO_PI 2.0 * 3.141592654
```

В дальнейшем вы можете использовать это имя в любом месте программы, где необходимо поставить значение 2.0*3.141592654. Поэтому вы можете заменить утверждение return в функции circumference из предыдущего листинга следующим утверждением.

```
return TWO_PI * r;
```

Где бы ни встретилось символическое имя в программе на языке C, оно всегда заменяется на то выражение, которое стоит в правой части утверждения #define для данного имени. То есть когда препроцессор языка C обнаруживает имя TWO_PI в утверждении return, приведенном выше, он заменяет это имя на то выражение, которое было определено в утверждении #define. В результате вместо имени TWO_PI будет подставлено выражение 2.0*3.141592654.

Именно поэтому нельзя ставить в конце утверждения #define точку с запятой, т.к. происходит буквальная замена имени тем выражением, которое расположено за ним. Если вы поставите точку с запятой, как в утверждении

```
#define PI 3.141592654;
```

то в выражении произойдет буквальная замена имени PI на выражение 3.141592654;

```
return 2.0 * PI * r;
```

и в результате вы получите следующее.

```
return 2.0 * 3.141592654; * r;
```

А это приведет к синтаксической ошибке при компиляции.

Все выражения для препроцессора не должны быть обязательно синтаксически правильными, но они должны составлять правильную конструкцию в том выражении, где будет произведена замена. Например, выражение

```
#define LEFT_SHIFT_8 << 8
```

является вполне правильным, хотя выражение, которое стоит справа от `LEFT_SHIFT_8`, не является синтаксически верным. Такую замену вы можете использовать в таких утверждениях, как

```
x = y LEFT_SHIFT_8;
```

где содержимое переменной `y` будет сдвинуто влево на 8 битов и результат присвоен переменной `x`. Вы без ограничений можете использовать и следующие или аналогичные им утверждения

```
#define AND &&  
#define OR ||
```

и затем писать выражения наподобие

```
if ( x > 0 AND x < 10 )  
...
```

или

```
if ( y == 0 OR y == value )  
...
```

Вы даже можете использовать утверждение `#define` для оператора равенства.

```
#define EQUALS ==
```

А затем написать следующее утверждение.

```
if ( y EQUALS 0 OR y EQUALS value )  
...
```

Таким образом, вполне реально исключить возможность ошибки, которую обычно допускают начинающие программисты, используя вместо оператора равенства (`==`) оператор присваивания (`=`), а также улучшить читабельность программы.

Хотя эти примеры и демонстрируют мощь директивы `#define`, вы должны четко понимать, что такое переопределение синтаксиса языка считается очень плохим стилем. При этом посторонним будет нелегко разобраться в вашей программе.

Но продолжим рассмотрение директивы `#define` и отметим такую особенность, как вложение символических имен, т.е. одно символическое имя может использоваться при определении другого символического имени. Например, выражения:

```
#define PI 3.141592654  
#define TWO_PI 2.0 * PI
```

являются полностью корректными. Имя `TWO_PI` использует предварительно определенное имя `PI` и при этом исчезает необходимость еще раз вводить число 3.141592654.

Если переставить эти утверждения

```
#define TWO_PI 2.0 * PI  
#define PI 3.141592654
```

то также не возникнет никаких проблем. Общее правило такое: вы можете использовать все символические имена, которые заданы в вашей программе, при определении нового символического имени. Осмысленное задание символических имен позволяет уменьшить количество комментариев в программе. Рассмотрим следующее выражение.

```
if ( year % 4 == 0 && year % 100 != 0 || year % 400 == 0 )
...

```

Вы должны вспомнить, что это выражение присутствовало в программах, уже рассмотренных в книге, когда производилась проверка относительно того, является ли год високосным. Теперь мы зададим следующее определение символического имени и подставим его в выражение `if`.

```
#define IS_LEAP_YEAR year % 4 == 0 && year % 100 != 0 \
|| year % 400 == 0
...
if ( IS_LEAP_YEAR )
...

```

Как видим, все вполне понятно и просто, а значит, нет никакой необходимости в дополнительных комментариях.

Обычно препроцессор предполагает, что определение содержится в одной строке программы. Если необходимо сделать перенос на другую строку, то последним символом в первой строке должен быть символ обратной черты (`\`). Этот символ свидетельствует только о продолжении определения, а сам символ при этом не учитывается. Таким образом можно использовать несколько строк, располагая в конце каждой строки символ обратной черты.

Еще раз отметим, что предыдущее утверждение `if` полностью отображает логику рассуждений и нет необходимости в дополнительных комментариях. В данном случае символическое имя `IS_LEAP_YEAR` можно рассматривать как функцию. Разумеется, вы можете использовать и функцию, чтобы достичь такой же степени читабельности программы. Здесь выбор всегда за вами. Разумеется, использование функции `is_leap_year` будет более универсальным решением, т.к. в этом случае можно использовать аргументы. Это позволит произвести проверку для любой переменной `year`, а не только для той, что будет задана в символическом имени, которое в данном случае ограничивает некоторые возможности. Но на самом деле вы можете писать такие символические имена, которые принимают аргументы, о чем мы поговорим в следующей главе.

Аргументы и макроопределения

Символическое имя `IS_LEAP_YEAR` вполне реально определено таким образом, чтобы с ним можно было использовать аргументы.

```
#define IS_LEAP_YEAR(y) y % 4 == 0 && y % 100 != 0 \
|| y % 400 == 0

```

В отличие от функции, здесь не задается тип аргумента `y`, поскольку просто выполняется буквальная подстановка текста и не выполняется проверок, как для функции. Также обратите внимание, что не должно быть пробела в утверждении `#define` между символическим именем и левой скобкой списка аргументов.

Используя предыдущее определение, можно записать утверждения следующим образом

```
if ( IS_LEAP_YEAR (year) )
...

```

с тем чтобы проверить, соответствует ли значение переменной `year` високосному году. Или можно записать утверждение следующим образом.


```
if ( IS_LEAP_YEAR (next_year) )
```

```
...
```

В предыдущем утверждении символическое имя `IS_LEAP_YEAR` будет буквально заменено в утверждении `if` благодаря значению, которое будет присвоено переменной `next_year`. При этом предыдущее значение переменной `y` уже не будет использоваться. Поэтому после подстановки в утверждении `if` будет рассчитываться следующее выражение.

```
if ( next_year % 4 == 0 && next_year % 100 != 0 \
    || next_year % 400 == 0 )
```

```
...
```

В языке программирования C подобный способ, с помощью которого задается символическое имя, часто называют *макροопределением*. Такое название обычно употребляется для определений, в которых используются аргументы. Преимущество использования макроопределения перед функциями заключается в том, что тип аргумента не контролируется. Например, рассмотрим макроопределение с именем `SQUARE`, которое просто вычисляет квадратный корень аргумента. Определение

```
#define SQUARE(x) x * x
```

позволяет впоследствии писать утверждения

```
y = SQUARE (v);
```

для присваивания переменной `y` значения v^2 . Независимо от того, какого типа будет переменная `v` (типа `int`, `long` или `float`), будет использоваться одно и то же макроопределение. Если же использовать для вычисления квадратного корня функцию `SQUARE`, которая принимает аргумент типа `int`, то невозможно будет рассчитать квадратный корень для переменной типа `double`.

Приведем еще одно замечание относительно использования макроопределений в программе. Так как выражение для вычисления макроопределения подставляется непосредственно в программу, то размер ее увеличится. Но при этом не затрачивается время на вызов функции, которая может использоваться вместо макроопределения. То есть использование макроопределений повышает быстродействие программы.

Правда, использование макроопределения `SQUARE` все-таки сопряжено с некоторыми сложностями, которые необходимо учитывать. Как уже говорилось, утверждение

```
y = SQUARE (v);
```

присваивает переменной `y` значение v^2 . А что произойдет, если написать утверждение следующим образом?

```
y = SQUARE (v + 1);
```

В этом случае для выражения $(v+1)$ на самом деле не будет вычисляться квадратный корень. Поскольку производится буквальная подстановка аргумента в макроопределение, то прероцессор преобразует эту запись следующим образом.

```
y = v + 1 * v + 1;
```

И это, как видим, не приводит к ожидаемому результату. Для исправления ситуации, при описании макроопределения `SQUARE` необходимо ставить круглые скобки.

```
#define SQUARE(x) ( (x) * (x) )
```

Хотя это и кажется несколько необычным, помните, что выражение, используемое в макроопределении, всегда подставляется буквально, без предварительного вычисления значения аргумента. Поэтому при использовании измененного выражения для макроопределения SQUARE, правая часть утверждения

```
y = SQUARE (v + 1);
```

будет заменена на вполне корректное выражение.

```
y = ( (v + 1) * (v + 1) );
```

Условные операторы также могут использоваться в макроопределениях. В следующем утверждении задается макроопределение с именем MAX, которое вычисляет максимальное значение из двух величин.

```
#define MAX(a,b) ( ((a) > (b)) ? (a) : (b) )
```

Это макроопределение можно использовать и таких утверждениях, как

```
limit = MAX (x + y, minValue);
```

где переменной limit будет присвоено максимальное значение при сравнении x+y и minValue. Выражение необходимо взять в круглые скобки для того, чтобы такие выражения, как

```
MAX (x, y) * 100
```

выполнялись правильно. А круглые скобки вокруг каждого аргумента гарантируют правильное вычисление следующих выражений.

```
MAX (x & y, z)
```

Поразрядный оператор AND имеет более низкий приоритет, чем оператор ">", используемый в макроопределении. В выражении без круглых скобок оператор ">" будет вычисляться перед оператором AND, что может привести к неправильному результату.

В следующем макроопределении производится проверка регистра символа

```
#define IS_LOWER_CASE(x) ( ((x) >= 'a') && ((x) <= 'z') )
```

и это позволяет писать следующие утверждения.

```
if ( IS_LOWER_CASE (c) )  
...
```

Вы даже можете использовать это макроопределение в макроопределении для преобразования символов в кодировке ASCII из строчных в заглавные, оставляя все заглавные символы без изменения.

```
#define TO_UPPER(x) ( IS_LOWER_CASE (x) ? (x) - 'a' + 'A' : (x) )
```

Цикл программы для решения этой задачи представлен ниже.

```
while ( *string != '\0' )  
{  
    *string = TO_UPPER (*string);  
    ++string;  
}
```

Здесь будут просмотрены все символы строки string и все строчные буквы будут преобразованы в заглавные.

Для этой же цели можно использовать и функции стандартной библиотеки для преобразования символов. Такие функции, как `islower` и `toupper`, выполняют то же самое, что и макроопределения `IS_LOWER_CASE` и `TO_UPPER`. Подробнее об этом рассказывается в приложении Б, "Стандартные библиотеки языка C".

Переменное число аргументов в макроопределении

Макроопределение можно задать и таким образом, что оно будет принимать неопределенное или переменное число аргументов. Об этом необходимо предупредить препроцессор с помощью трех точек в конце списка аргументов. На эти аргументы можно сделать ссылку с помощью специального идентификатора `__VA_ARGS__`. Например, задавая макроопределение с именем `debugPrintf`, следует использовать переменное число аргументов.

```
#define debugPrintf(...) printf ("DEBUG: " __VA_ARGS__ );
```

Корректное использование макроопределения можно представить как

```
debugPrintf ("Hello world!\n");
```

или как

```
debugPrintf ("i = %i, j = %i\n", i, j);
```

В первом случае будет сделан следующий вывод

```
DEBUG: Hello world!
```

А во втором случае, например, если значение переменной `i` будет равно 100, а значение переменной `j` — 200, то вывод будет следующим.

```
DEBUG: i = 100, j = 200
```

Вызов функции `printf` в первом случае будет изменен препроцессором на

```
printf ("DEBUG: " "Hello world\n");
```

который также может объединить соседние строковые константы. Поэтому окончательный вариант функции `printf` будет выглядеть следующим образом.

```
printf ("DEBUG: Hello world\n");
```

Оператор

Если разместить оператор `#` перед параметром в макроопределении, препроцессор создаст строковую константу из параметра при вызове макроопределения. Например, задав макроопределение как

```
#define str(x) # x
```

и сделав вызов макроопределения следующим образом

```
str (testing)
```

получим строку

```
"testing"
```

которую создал препроцессор. Поэтому следующий вызов функции `printf`

```
printf (str (Programming in C is fun.\n));
```

будет эквивалентен вызову

```
printf ("Programming in C is fun.\n");
```

Препроцессор просто расставляет двойные кавычки вокруг аргумента. Двойные кавычки или обратная черта в аргументах препроцессором сохраняются. Поэтому вызов:

```
str ("hello")
```

преобразуется в

```
"\"hello\""
```

Еще один пример использования оператора # показан в следующем макроопределении.

```
#define printint(var) printf (# var " = %i\n", var)
```

Это макроопределение используется для отображения значений целочисленной переменной. Если переменная `count` является целочисленной переменной со значением 100, то утверждение

```
printint (count);
```

будет преобразовано следующим образом.

```
printf ("count" " = %i\n", count);
```

А после того как будет сделано объединение соседних строк, вызов функции будет выглядеть так, как показано ниже.

```
printf ("count = %i\n", count);
```

Таким образом оператор # дает возможность создавать строковые константы для аргумента макроопределения.

Осталось уточнить, что пробел между оператором # и именем параметра не является обязательным.

Оператор

Этот оператор используется в макроопределениях для объединения двух лексем. Он предшествует (или следует) имени параметра макроопределения. Препроцессор обрабатывает аргументы при вызове макроопределения и создает одну лексему из двух, между которыми находится оператор ##.

Предположим, что необходимо создать список переменных от `x1` до `x100`. Вы можете написать макроопределение с именем `printx`, в который в качестве аргументов будут передаваться целые значения от 1 до 100 и затем отображаться вместе с символом "x".

```
#define printx(n) printf ("%i\n", x ## n)
```

Выражение `x##n`, в котором между символом "x" и аргументом `n` находится оператор ##, будет преобразовано в одну лексему. Поэтому вызов

```
printx (20);
```

будет преобразован в следующее выражение.

```
printf ("%i\n", x20);
```

Макроопределение `printx` может даже использовать ранее определенное макроопределение `printint` для получения имени переменной при ее отображении на экране.

```
#define printx(n) printint(x ## n)
```

Следующий вызов

```
printx (10);
```

сначала будет преобразован в выражение

```
printint (x10);
```

а затем в выражение

```
printf ("x10" " = %i\n", x10);
```

и после объединения строк в окончательное выражение

```
printf ("x10 = %i\n", x10);
```

Утверждение `#include`

Имея уже некоторый опыт программирования на языке C, вы обнаружите, что вы собрали целую коллекцию макроопределений, которые вы обычно используете в каждой из своих программ. И вместо того, чтобы набирать их в каждой новой программе, вы можете собрать их в отдельном файле, с тем чтобы подключать его к очередной программе с помощью директивы `#include`. Препроцессор будет обрабатывать такие файлы, которые имеют расширение `.h` и называются *заголовочными* или *подключаемыми* файлами.

Предположим, вы написали несколько программ для выполнения различных метрических преобразований. И наверное, вам захочется задать с помощью символических имен некоторые константы, которые будут использоваться в расчетах.

```
#define INCHES_PER_CENTIMETER 0.394
#define CENTIMETERS_PER_INCH 1 / INCHES_PER_CENTIMETER
#define QUARTS_PER_LITER 1.057
#define LITERS_PER_QUART 1 / QUARTS_PER_LITER
#define OUNCES_PER_GRAM 0.035
#define GRAMS_PER_OUNCE 1 / OUNCES_PER_GRAM
...
```

Предположим также, что вы записали эти строки в отдельный файл с именем `metric.h`. Впоследствии любая программа, в которой необходимо использовать константы, описанные в файле `metric.h`, может очень легко получить к ним доступ с помощью директивы препроцессора.

```
#include "metric.h"
```

Причем это утверждение должно появиться еще до того, как будет использовано символическое имя из файла `metric.h`. Обычно оно ставится в начале программы. Препроцессор ищет в системе указанный файл и копирует его содержимое в программу в том месте, где встретилась директива `#include`. Поэтому любое определение из

файла можно рассматривать так, как будто оно непосредственно написано в программе в указанном месте.

Двойные кавычки вокруг имени файла обязывают препроцессор производить поиск файла в одном или нескольких каталогах, и первым из них обычно будет тот каталог, в котором находится исходный файл программы. То, каким образом в дальнейшем производится поиск, зависит от системы. Если файл в данном каталоге не найден, препроцессор автоматически переходит к поиску в другом системном каталоге.

Когда имя файла заключено в угловые скобки ("`<`" "`>`"), например, следующим образом

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

препроцессор производит поиск подключаемого файла в специально используемом для этого каталоге (или каталогах). Как и ранее, расположение этих каталогов зависит от системы. В операционной системе Unix (включая систему Mac OS X), каталог для подключаемых файлов будет иметь путь `/usr/include`, поэтому стандартный подключаемый файл `stdio.h` будет находиться по следующему адресу.

```
/usr/include/stdio.h.
```

Для того чтобы увидеть на реальном примере, как используются подключаемые файлы, запишите шесть ранее приведенных определений символических имен в файл с именем `metric.h`, затем наберите и запустите на выполнение программу из листинга 13.3.

Листинг 13.3. Использование директивы `#include`

/ В программе демонстрируется использование директивы `#include`.*

*Замечание: в программе предполагается, что определения символических имен записаны в файле с именем `metric.h` */*

```
#include <stdio.h>
#include "metric.h"
int main (void)
{
    float liters, gallons;

    printf ("*** Liters to Gallons ***\n\n");
    printf ("Enter the number of liters: ");
    scanf ("%f", &liters);
    gallons = liters * QUARTS_PER_LITER / 4.0;
    printf ("%g liters = %g gallons\n", liters, gallons);

    return 0;
}
```

Листинг 13.3. Вывод

```
*** Liters to Gallons ***
Enter the number of liters: 55.75
55.75 liters = 14.73 gallons.
```

Предыдущий пример довольно простой, т.к. в нем из файла `metric.h` используется только одно символическое имя `QUARTS_PER_LITER`. Тем не менее, этот пример весьма

показательный, так как после записи этого символического имени в файл `metric.h`, его можно использовать в любой программе, в которой будет записана соответствующая директива `#include`.

Идея использования подключаемых файлов хороша еще и потому, что можно сгруппировать все необходимые определения в одном месте и быть уверенным, что ничего не будет случайно пропущено. Более того, если вдруг будет замечена ошибка, то ее легко исправить в одном месте и не надо производить поиск по многим файлам. Все программы, которые использовали ошибочное значение, не надо будет исправлять, а только заново перекомпилировать.

Вы можете помещать в подключаемый файл все что хотите, а не только директивы `#define`. Использование подключаемых файлов для размещения всех препроцессорных определений, описаний структур, прототипов объявлений и объявления глобальных переменных, является признаком хорошего стиля программирования.

К сказанному выше о подключаемых файлах следует добавить, что они могут быть вложенными. Это означает, что в одном подключаемом файле можно производить ссылку на другой подключаемый файл, в котором есть ссылка на третий подключаемый файл и т.д.

Системные подключаемые файлы

Ранее уже упоминалось о том, что стандартный подключаемый файл `stddef.h` содержит определение для символического слова `NULL`, которое часто используется при проверке значения указателей. В начале этой главы говорилось о том, что стандартный заголовочный файл `math.h` содержит определение для символического слова `M_PI`, которое используется вместо числового значения для π .

Стандартный подключаемый файл `stdio.h` содержит информацию о подпрограммах ввода-вывода, находящихся в стандартных библиотеках ввода-вывода. Этот подключаемый файл подробно описан в главе 16, "Операции ввода-вывода в языке C". Вы должны будете подключить этот файл, когда будете использовать подпрограммы ввода-вывода в своей программе.

Два других полезных стандартных подключаемых файла называются `limits.h` и `float.h`. В первом из них (`limits.h`) содержатся зависимые от системы значения, которые определяют размеры различных символьных и целочисленных типов данных. Например, максимальный размер для типа `int` скрывается за символическим именем `INT_MAX`. А для максимального размера типа `unsigned long int` выбрано символическое имя `ULONG_MAX`, и т.д.

Подключаемый файл `float.h` предоставляет информацию о вещественных типах данных. Например, символическое имя `FLT_MAX` определяет максимальное вещественное число, а `FLT_DIG` определяет число десятичных цифр, с тем чтобы задать необходимую точность вещественных (`float`) чисел.

Другие подключаемые файлы содержат прототипы объявлений для различных функций, сохраняемых в системных библиотеках. Например, подключаемый файл `string.h` содержит прототипы объявлений для библиотеки подпрограмм, которая используется для выполнения операций со строками, такими как копирование, сравнение или объединение.

Более подробные сведения о подключаемых файлах представлены в приложении Б.

Условная компиляция

Препроцессор языка программирования C предлагает и такие возможности, как *условная компиляция*. Условная компиляция часто используется для создания программ, которые должны компилироваться и запускаться на различных компьютерных системах. Ее также часто используют для переключения между различными утверждениями в программе, такими как утверждения для отладки, которые выводят на печать значения, используемые при режиме трассировки.

Утверждения `#ifdef`, `#endif`, `#else` и `#ifndef`

Ранее в этой главе уже было показано, как можно улучшить функцию `rotate` из главы 12, с тем чтобы она стала переносимой. Вы имели возможность убедиться, что использование символического имени помогло решить эту проблему. Определение

```
#define kIntSize 32
```

использовалось для того, чтобы задать в одном месте количество битов, содержащихся в типе `unsigned int`. Но, как не раз отмечалось, для полного исключения этой зависимости необходимо, чтобы программа самостоятельно определяла количество битов, используемых системой для типа `unsigned int`.

К сожалению, иногда программа должна полагаться на зависимые от системы параметры — имена файлов, которые могут задаваться по-разному в различных системах, в том числе операционных.

Если вы создадите большую программу, которая будет зависеть от аппаратной или программной составляющей компьютерной системы (обычно эту зависимость стараются минимизировать), вы должны по окончании работы выделить те значения, которые могут изменяться при переносе программы на различные компьютерные системы.

Вы можете устранить такие проблемы, имея возможность изменить эти значения, когда программа переносится на другой компьютер. Все изменения могут быть выполнены автоматически с использованием возможностей условной компиляции препроцессора.

В качестве примера рассмотрим следующие утверждения.

```
#ifdef UNIX
# define DATADIR "/uxnl/data"
#else
# define DATADIR "\\usr\data"
#endif
```

Здесь настройка заключается в определении символического имени `DATADIR`, которому присваивается значение `"/uxnl/data"`, если раньше было определено символическое имя `UNIX`, в противном случае имени `DATADIR` присваивается значение `"\\usr\data"`. Как видим, в данном случае можно ставить один или несколько пробелов после символа `#`, с которого начинается утверждение препроцессора.

Поведение утверждений `#ifdef`, `#else` и `#endif` вполне предсказуемо. В зависимости от того, определено или нет символическое имя, используемое с директивой `#ifdef`, препроцессором выполняются или игнорируются директивы `#else`, `#elif` или `#endif`.

Для того чтобы определить символическое имя `UNIX`, нужно использовать утверждение

```
#define UNIX 1
```


или даже более простое следующее утверждение.

```
#define UNIX
```

Большинство компиляторов позволяют определить имя при компиляции программы с помощью специальной опции в командной строке. Например, для компилятора gcc командная строка должна быть следующей.

```
gcc -D UNIX program.c
```

Здесь для препроцессора определяется имя UNIX, заставляя все утверждения `#ifndef UNIX` внутри программы `program.c` принимать значение TRUE (обратите внимание, что в командной строке опция `-D UNIX` должна находиться перед именем программы). Такая техника определения имен позволяет компилировать программу без редактирования исходного файла.

Некоторое значение также можно присвоить символическому имени прямо в командной строке. Например, команда

```
gcc -D GNUDIR=/c/gnustep program.c
```

вызывает компилятор gcc, задавая для имени GNUDIR значение `/c/gnustep`.

Множественное включение подключаемых файлов

Утверждение `#ifndef` используется так же, как и утверждение `#ifdef`, за исключением того, что последующие утверждения будут выполняться лишь в том случае, если используемое с ним символическое имя не определено. Это во многих случаях позволяет избежать множественного включения подключаемых файлов, например, внутри самих подключаемых файлов. Как известно, необходимо, чтобы подключаемый файл не встречался несколько раз. Следовательно, если вы хотите быть уверены, что не произойдет множественного подключения, вы должны определить для файла уникальное символическое имя, которое впоследствии и будете контролировать. Рассмотрим последовательность утверждений.

```
#ifndef _MYSTDIO_H
#define _MYSTDIO_H
...
#endif /* _MYSTDIO_H */
```

Предположим, вы написали эти строки в файле с именем `mystdio.h`. Если вы теперь подключите этот файл в вашу программу с помощью следующего утверждения

```
#include "mystdio.h"
```

то внутри файла будет произведена проверка утверждения `#ifndef` с целью определить символическое имя `_MYSTDIO_H`. Поскольку при первом подключении оно еще не определено, то строки между директивами `#ifndef` и `#endif` будут включены в программу. Здесь можно поместить все утверждения, которые вы хотите включить в программу из подключаемого файла. Обратите внимание на то, что на следующей строке подключаемого файла находится определение символического имени `_MYSTDIO_H`. При повторной попытке подключить этот файл к программе символическое имя `_MYSTDIO_H` уже будет определено и, соответственно, все последующие утверждения, вплоть до `#endif`, которое обычно помещается в конце файла, будут пропущены, что и позволит избежать множественного включения.

Утверждения препроцессора #if и #elif

Утверждение препроцессора #if позволяет использовать более общий способ условной компиляции. Утверждение #if может использоваться для проверки равенства константного выражения нулю. Если выражение не равно нулю, последующие строки, вплоть до #else, #elif или #endif, выполняются, а в противном случае они пропускаются. Предположим, например, что вы определили символическое имя OS, которое будет соответствовать 1, если используется операционная система Macintosh OS; 2, — если используется Windows; 3 — если используется операционная система Linux и т.д. Вы можете написать следующую последовательность утверждений условной компиляции программы для соответствующей операционной системы.

```
#if OS == 1 /* Mac OS */
...
#elif OS == 2 /* Windows */
...
#elif OS == 3 /* Linux */
...
#else
...
#endif
```

Для большинства компиляторов вы можете присвоить значение для имени OS в командной строке с помощью опции -D, о чем говорилось ранее. Командная строка.

```
gcc -D OS=2 program.c
```

заставит выполнить компиляцию программы program.c, используя для имени OS значение 2. Это приведет к тому, что программа будет скомпилирована для работы в операционной системе Windows.

Специальный оператор

```
defined (name)
```

может также использоваться в утверждениях #if. Следующие препроцессорные утверждения:

```
#if defined (DEBUG)
...
#endif
```

и

```
#ifdef DEBUG
...
#endif
```

делают одно и то же. Утверждения

```
#if defined (WINDOWS) || defined (WINDOWSNT)
# define BOOT_DRIVE "C:/"
#else
# define BOOT_DRIVE "D:/"
#endif
```

задают символическое имя BOOT_DRIVE как "C: /", если определены имена WINDOWS или WINDOWSNT, и как "D: /" — в противном случае.

Утверждение `#undef`

В некоторых случаях может потребоваться снять определение с уже определенного имени. Это можно сделать с помощью утверждения `#undef`. Для того чтобы сделать отдельное имя неопределенным, необходимо написать следующее.

```
#undef name
```

Поэтому утверждение

```
#undef WINDOWS_NT
```

приведет к тому, что символическое имя `WINDOWS_NT` станет неопределенным. Последующие выражения `#ifdef WINDOWS_NT` или `#if defined (WINDOWS_NT)` будут возвращать значение `FALSE`.

На этом завершается обсуждение препроцессора. У вас была возможность убедиться, что с помощью препроцессора можно сделать программу более наглядной и легче понимаемой. Ее легче писать и модифицировать. Вы также узнали о том, как можно использовать подключаемые файлы с целью сгруппировать отдельные определения и объявления, которые могут быть разбросаны по всем программам. Дополнительные препроцессорные утверждения, которые не описаны в этой главе, можно найти в приложении А, "Краткое изложение языка программирования С".

В следующей главе вы узнаете больше о типах данных и о преобразовании типов.

Упражнения

1. Наберите и запустите на выполнение все программы данной главы, не забывая подключить необходимые файлы для программы из листинга 13.3. Сравните полученные результаты с теми, что приведены для каждой программы в книге.
2. Найдите стандартные подключаемые файлы `stdio.h`, `limits.h` и `float.h` на вашем компьютере (для системы Unix см. каталог `/usr/include`). Ознакомьтесь с их содержимым.
3. Напишите макроопределение с именем `MIN` для получения наименьшего из двух значений. Затем напишите программу для проверки этого макроопределения.
4. Напишите макроопределение с именем `MAX3`, которое возвращает максимальное значение из трех величин. Затем напишите программу для проверки этого макроопределения.
5. Напишите макроопределение с именем `SHIFT`, которое выполняет задачи, идентичные функции `shift` из листинга 12.3.
6. Напишите макроопределение с именем `IS_UPPER_CASE`, которое возвращает ненулевое значение, если принимаемый символ введен в верхнем регистре.
7. Используйте макроопределение `IS_LOWER_CASE`, описанное в данной главе, и макроопределение `IS_UPPER_CASE` из упражнения 6.
8. Напишите макроопределение с именем `IS_DIGIT`, которое возвращает ненулевое значение, если принимаемый символ является цифрой от 0 до 9. Используйте это макроопределение для написания другого макроопределения с именем `IS_SPECIAL`, которое возвращает не нулевое значение, если принимаемый символ считается специальным символом, т.е. он не является цифрой или буквой. Используйте макроопределение, разработанное в упражнении 7.

9. Напишите макроопределение с именем `ABSOLUTE_VALUE`, которое рассчитывает абсолютное значение числового аргумента. Убедитесь, что такое выражение, как

```
ABSOLUTE_VALUE (x + delta)
```

рассчитывается правильно.

10. Рассмотрите макроопределение `printint` из данной главы.

```
#define printint(n) printf ("%i\n", x ## n)
```

Можно ли его использовать для отображения 100 переменных `x1-x100`?

Следующие утверждения будут правильными?

Да или нет? Почему?

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

11. Проверьте функции стандартных библиотек, которые выполняют те же задачи, что и макроопределения из упражнений 6, 7 и 8. Эти функции называются `isupper`, `isalpha` и `isdigit`. Для проверки их работы необходимо подключить в вашу программу стандартный файл `ctype.h`, с тем чтобы ими можно было пользоваться.

Еще о типах данных

В этой главе будет представлен пока новый для вас *перечислимый* тип данных. Вы также познакомитесь с ключевым словом `typedef`, которое позволит присваивать выбранные вами имена для основных типов данных или для унаследованных от них типов. Наконец, в этой главе вы узнаете о способах, с помощью которых можно задать точность при преобразованиях типов данных в выражениях.

Перечислимые типы данных

Как известно, очень ценной и известной считается возможность объявлять переменную и задавать для нее все допустимые значения, которые могут сохраняться в переменной. Например, предположим, что у вас есть переменная с именем `myColor` и вы хотите использовать ее для хранения только одного из основных цветов: `red` (красный), `yellow` (желтый) или `blue` (синий). Это легко можно сделать с помощью перечислимого типа данных.

Перечислимый тип данных задается при объявлении с помощью ключевого слова `enum`. Непосредственно за этим словом должно находиться имя перечислимого типа данных с последующим списком идентификаторов, заключенных в фигурные скобки. Идентификаторы определяют те значения, которые могут быть присвоены данному типу. Например, в утверждении

```
enum primaryColor { red, yellow, blue };
```

объявляется тип данных `primaryColor`. Переменным, объявленным как тип `primaryColor`, в программе можно присвоить только значения `red`, `yellow` и `blue`. Других значений им присваивать нельзя. Но так должно быть теоретически. При попытке присвоить им другое значение некоторые компиляторы зафиксируют ошибку, а некоторые могут и не обратить на это никакого внимания.

При объявлении переменной как тип `enum primaryColor`, необходимо вновь использовать ключевое слово `enum` с последующим именем перечислимого типа, за которым следует список переменных. Поэтому в утверждении

```
enum primaryColor myColor, gregsColor;
```

объявляются две переменные `myColor` и `gregsColor`, которые будут иметь тип `primaryColor`. Допустимыми значениями, которые могут быть присвоены этим

переменным, являются имена `red`, `yellow` и `blue`. Поэтому вполне допустимыми являются такие утверждения, как

```
myColor = red;

и

if ( greg'sColor == yellow )
...

```

В качестве еще одного примера использования перечислимого типа данных, зададим тип `enum month`. Переменным этого типа можно присваивать только названия месяцев года.

```
enum month { january, february, march, april, may, june,
july, august, september, october, november, december };

```

В действительности компилятор языка программирования C трактует эти названия как целочисленные константы. Начиная с первого имени в списке компилятор присваивает именам последовательные целочисленные значения. Первое имя имеет значение 0. Если в программе содержатся строки наподобие

```
enum month thisMonth;
...
thisMonth = february;

```

то значение 1 присваивается переменной `thisMonth` (а не имя `february`), поскольку это второе имя в списке идентификаторов.

Если вы хотите получить для отдельного имени специально выбранное целочисленное значение, то вы можете присвоить это значение непосредственно при объявлении типа данных. Все последующие целочисленные значения будут присваиваться автоматически, начиная с заданного значения и увеличиваясь на 1 для очередного идентификатора. Например, при объявлении

```
enum direction { up, down, left = 10, right };

```

перечислимый тип данных `direction` будет использовать имена `up`, `down`, `left` и `right`. Компилятор присвоит имени `up` значение 0, поскольку это первое имя в списке. Для второго имени `down` будет присвоено значение 1, а следующему имени `left` будет присвоено значение 10, поскольку было выполнено явное присваивание при объявлении. Последнему имени `right` будет присвоено значение 11, поскольку в списке оно идет непосредственно за именем `left`.

В программе из листинга 14.1 показано простое использование перечислимого типа данных. В перечислимом типе данных `month` для имени `january` задается значение 1, поскольку этот месяц является первым месяцем года, а нумерация месяцев производится с единицы. В программе считывается номер месяца и возвращается название месяца, соответствующее введенному номеру. Мы только что упомянули о том, что все заданные в списке идентификаторы трактуются как целочисленные значения, и поэтому эти значения можно использовать в конструкции `case`. Переменной `days` присваиваются значения, которые соответствуют количеству дней в каждом месяце, и эти значения отображаются после названий месяцев. Для определения количества дней в феврале выводится специальное пояснение.

Листинг 14.1. Использование перечислимых типов данных

```
// Программа выводит количество дней для каждого месяца
#include <stdio.h>
int main (void)
{
    enum month { january = 1, february, march, april, may, june,
    july, august, september, october, november, december };
    enum month aMonth;
    int days;

    printf ("Enter month number: ");
    scanf ("%i", &aMonth);
    switch (aMonth )
    {
        case january:
        case march:
        case may:
        case july:
        case august:
        case october:
        case december:
            days = 31;
            break;
        case april:
        case june:
        case september:
        case november:
            days = 30;
            break;
        case february:
            days = 28;
            break;
        default:
            printf ("bad month number\n");
            days = 0;
            break;
    }
    if ( days != 0 )
        printf ("Number of days is %i\n", days);
    if ( aMonth == february )
        printf ("...or 29 if it's a leap year\n");

    return 0;
}
```

Листинг 14.1. Вывод

```
Enter month number: 5
Number of days is 31
```

Листинг 14.1. Вывод (Повторение)

```
Enter month number: 2
Number of days is 28
...or 29 if it's a leap year
```

В списке для идентификаторов перечислимых типов можно указывать одинаковые значения.

```
enum switch { no=0, off=0, yes=1, on=1 };
```

Идентификаторам `no` и `off` для типа `enum switch` будет присвоено значение 0, а идентификаторам `yes` и `on` будет присвоено значение 1.

Явно присвоить целочисленное значение идентификатора из списка можно с помощью оператора приведения типов. Так, если переменная `monthValue` является целочисленной переменной, которая имеет значение 6, то, например, выражение

```
thisMonth = (enum month) (monthValue - 1);
```

будет вполне допустимым и присвоит значение 5 переменной `thisMonth`.

При написании программ с использованием перечислимых типов данных, старайтесь не использовать тот факт, что значения перечислимых типов можно трактовать как целые числа. Старайтесь относиться к ним как к особым типам данных. Перечислимые типы дают вам возможность использовать символические имена в качестве целых чисел. Если возникнет необходимость в изменении значений этих имен, то лучше всего это делать в том месте, где объявляется перечислимый тип. При непродуманном использовании символических имен и их значений можно потерять все преимущества от использования перечислимых типов.

При объявлении перечислимого типа данных возможны варианты, подобные вариантам при объявлении структур. Указание типа данных может быть пропущено, а переменная может быть объявлена как описанный перечислимый тип. Например, рассмотрим утверждение

```
enum { east, west, south, north } direction;
```

где объявляется перечислимый тип данных без имени со значениями `east`, `west`, `south` и `north`, и одновременно объявляется переменная `direction`, которая будет соответствовать этому типу.

Поведение перечислимого типа в программе подобно поведению структуры, и объявленные переменные имеют такую область видимости, которая соответствует с ранее описанным правилам. Объявление перечислимого типа в блоке ограничивает область видимости перечислимого типа этим блоком. С другой стороны, объявление перечислимого типа в начале программы, за пределами всех функций, делает это объявление глобальным в файле.

При объявлении перечислимого типа данных необходимо убедиться в том, что имена в списке являются уникальными по отношению к другим именам переменных, заданных в этой же области видимости.

Утверждение `typedef`

Язык программирования C позволяет создавать альтернативные имена для типов данных. Это можно сделать с помощью ключевого слова `typedef`. Так, в утверждении

```
typedef int Counter;
```

объявляется тип `Counter`, который после этого объявления можно использовать наравне с ключевым словом `int` и применять его для объявления целочисленной переменной. То есть для объявления переменной можно использовать тип `Counter`, как показано в следующем утверждении

```
Counter j, n;
```

Компилятор будет трактовать объявления переменных `j` и `n` из предыдущего примера как объявления целочисленных переменных. Главное преимущество от использования альтернативного типа данных, т.е. от использования ключевого слова `typedef`, заключается в дополнительном пояснении роли переменных, которые несут определенную смысловую нагрузку и этим способствуют повышению читабельности программы. Так, например, будет более понятна роль и назначение переменных `j` и `n`, которые объявлены как тип `Counter` (Счетчик). Использование же при объявлении ключевого слова `int` не внесет никакой дополнительной ясности о назначении переменных. Разумеется, использование осмысленных имен переменных будет также способствовать более эффективному восприятию программы.

Во многих случаях утверждение `typedef` может рассматриваться как эквивалентное для утверждения `#define`, а значит, может быть им заменено. Например, вместо предыдущего объявления можно использовать следующее:

```
#define Counter int
```

При этом будет получен тот же самый результат. Но поскольку утверждение `typedef` обрабатывается непосредственно компилятором языка C, а не препроцессором, это утверждение обеспечивает большую гибкость, чем утверждение `#define`, и его необходимо использовать, если приходится присваивать имена унаследованным типам данных. Например, следующее утверждение `typedef`

```
typedef char Linebuf [81];
```

создаст тип с именем `Linebuf`, который будет представлять массив из 81 символа. При последовательном объявлении переменных типа `Linebuf`, как показано ниже

```
Linebuf text, inputLine;
```

будет понятно, что создаются массивы, содержащие по 81 символу. Эквивалентное объявление можно сделать следующим образом.

```
char text[81], inputLine[81];
```

Но в этом случае имя `Linebuf` не будет выполнять ту же роль, если его задать с помощью препроцессорного утверждения `#define`.

В следующем утверждении `typedef` задается тип с именем `StringPtr`, который будет являться указателем на тип `char`.

```
typedef char *StringPtr;
```

Последующее объявление переменных типа `StringPtr` наподобие

```
StringPtr buffer;
```

будет понято компилятором C как объявление указателей на символ.

Для задания нового типа с помощью ключевого слова `typedef`, выполните следующие шаги.

1. Напишите утверждение в качестве объявления переменной заданного типа.
2. Замените используемое имя переменной на новое имя типа.
3. Затем перед этими словами поставьте ключевое слово `typedef`.

Как пример использования этого правила, зададим тип с именем `Date`, который должен определять структуру с тремя целочисленными переменными с именами

month, day и year. Для этого сначала необходимо объявить переменную, т.е. описать структуру и разместить за ней (перед точкой с запятой) имя переменной. Этим именем в нашем случае будет Date. Затем подставим перед описанием структуры ключевое слово typedef.

```
typedef struct
{
    int month;
    int day;
    int year;
} Date;
```

После того как подобным образом задан новый тип, можно объявлять переменные нового типа Date.

```
Date birthdays[100];
```

При этом будет объявлен массив birthdays, содержащий 100 структур типа Date.

При работе с программой, исходный код которой располагается более чем в одном файле (см. главу 15, “Работа с большими программами”), то будет очень удобно размещать утверждения typedef в отдельном файле и подключать его к исходным файлам с помощью утверждения #include.

Рассмотрим еще один пример. Предположим, вы работаете над графическим пакетом, который должен рисовать окружности, линии, кривые и т.д. При этом вам придется много работать с системой координат. Поэтому удобно использовать утверждение typedef, в котором будет описана структура, содержащая два члена типа float: x и y, для которой используется символическое имя Point.

```
typedef struct
{
    float x;
    float y;
} Point;
```

После такого объявления вы можете более эффективно продолжать разработку графической библиотеки, используя все преимущества использования типа Point. Например, в следующем объявлении

```
Point origin = { 0.0, 0.0 }, currentPoint;
```

создаются две переменные origin и currentPoint типа Point, причем для членов первой переменной устанавливаются нулевые значения.

Ниже описана функция distance, в которой производится расчет расстояния между двумя точками.

```
#include <math.h>
double distance (Point p1, Point p2)
{
    double diffx, diffy;
    diffx = p1.x - p2.x;
    diffy = p1.y - p2.y;
    return sqrt (diffx * diffx + diffy * diffy);
}
```

Как уже отмечалось, функция `sqrt` является функцией стандартной библиотеки и используется для вычисления квадратного корня. Для использования этой функции необходимо подключить файл `math.h`, для чего используется директива компилятора `#include`. Помните, что утверждение `typedef` на самом деле не определяет новый тип, а только новое имя типа. Поэтому переменные `j` и `n` типа `Counter`, используемые в начале этого раздела, необходимо трактовать как переменные стандартного типа `int`. Именно так их и понимает компилятор языка программирования C.

Приведение типов

В главе 4, “Переменные, типы данных и арифметические выражения”, кратко уже упоминалось о том, что приведение типов может производиться неявно при вычислении некоторых выражений.

В качестве примера использовались типы данных `float` и `int`. Вы могли убедиться, что операции с типами `float` и `int` выполнялись как операции с вещественными числами, т.е. тип `int` был автоматически преобразован в тип `float`.

Вы также могли видеть, что можно явно использовать оператор приведения типов для преобразования одного типа в другой. Поэтому в утверждении

```
average = (float) total / n;
```

значение переменной `total` преобразовывается в тип `float` еще до того, как выполняется операция деления. При этом гарантируется, что деление будет выполняться по правилам вычислений с вещественными числами.

В языке программирования C используются четкие правила при вычислении выражений, состоящих из разных типов данных. Ниже приводится список правил, которые последовательно применяются при выполнении операций над двумя операндами в выражении.

1. Если один из операндов имеет тип `long double`, то другой операнд будет преобразован к типу `long double` и результирующее значение также будет иметь этот тип.
2. Если один из операндов имеет тип `double`, то другой операнд будет преобразован к типу `double` и результирующее значение также будет иметь этот тип.
3. Если один из операндов имеет тип `float`, то другой операнд будет преобразован к типу `float` и результирующее значение также будет иметь этот тип.
4. Если один из операндов имеет тип `_Bool`, `char`, `short int`, `bit field` или перечислимый тип, то они будут преобразованы к типу `int`.
5. Если один из операндов имеет тип `long long int`, то другой операнд будет преобразован к типу `long long int` и результирующее значение также будет иметь этот тип.
6. Если один из операндов имеет тип `long int`, то другой операнд будет преобразован к типу `long int` и результирующее значение также будет иметь этот тип.
7. Если достигнут этот пункт, то оба операнда имеют тип `int` и результирующее значение также будет иметь этот тип.

Представленная последовательность шагов является простейшим вариантом приведения операндов в выражениях. Правила будут более сложными при использовании в выражениях беззнаковых операндов.

Для ознакомления с полным списком правил обратитесь к приложению А, “Краткий обзор языка программирования С”. Следует помнить, что эти правила используются по умолчанию и для того, чтобы узнать тип результата, надо следовать этим правилам.

Как пример неявного приведения типов в соответствии с данными правилами, рассмотрим, как выполняется нижеприведенное выражение, где переменная *f* объявлена как тип *float*, переменная *i* — как *int*, переменная *l* — как *long int* и переменная *s* как *short int*.

```
f * i + l / s
```

Сначала рассмотрим перемножение *f* на *i*, которое является перемножением операндов типа *float* и *int*. Выполняя шаг 3, мы заметим, что если один операнд (*f*) является типом *float*, то другой операнд (*i*) должен быть приведен к типу *float*, а результат перемножения также будет иметь тип *float*.

Затем должно производиться деление операндов *l* by *s*, где делится тип *long int* на тип *short int*. В соответствии с шагом 4, операнд типа *short int* должен быть преобразован к типу *int*. Далее в соответствии с пунктом 6 второй операнд (*s*) должен быть приведен к типу *long int*, поскольку первый операнд (*l*) имеет тип *long int*. Таким образом, после выполнения операции деления будет получен результат типа *long int*. Дробная часть будет отброшена в соответствии с правилами деления целых чисел.

Наконец, будет выполнено сложение операнда типа *float* (как результат перемножения *f* на *i*) и операнда типа *long int*. В соответствии с правилом 3, если один из операндов имеет тип *float*, то другой операнд должен быть преобразован к этому типу. Окончательный результат, полученный при вычислении этого выражения, будет иметь тип *float*.

Вспомните, что приведение типов всегда можно выполнить явно для получения необходимого результата. Правда, он не всегда будет таким, как ожидалось, если полагаться на приведение типов по умолчанию.

Итак, если вы хотите получить и дробную часть в результате деления *l* на *s* в предыдущем выражении, то вы должны явно привести один из операндов к типу *float*, т.е. получая результат согласно правилам деления вещественных чисел. То есть необходимо написать следующее.

```
f * i + (float) l / s
```

В этом выражении операнд *l* будет преобразован в тип *float* до того, как будет выполнена операция деления, так как оператор преобразования типов имеет высший приоритет по отношению к оператору деления. После этого при делении будет выполнено автоматическое приведение второго операнда (*s*) к типу *float*, который и будет типом полученного результата.

Знаковое расширение

Когда типы *signed int* или *signed short int* приводятся к типам большего размера, знаковый бит копируется во все биты слева (знаковое расширение). Это необходимо для того, чтобы тип *short int*, имеющий, например, значение -5 , после приведения к типу *int* имел тоже значение -5 . Однако при приведении беззнаковых типов к типу большего размера, такое предполагаемое копирование знакового бита не осуществляется.

В некоторых системах, таких как Mac G4/G5 и в системах с процессорами Pentium, символы трактуются как значения со знаком. То есть, когда символ приводится к целочисленному типу, то должно происходить знаковое расширение. Поскольку стандартные символы принадлежат набору символов из таблицы кодов ASCII, то такое приведение не создает никаких проблем. Но если используется символьное значение не из стандартного набора, то необходимо учитывать возможные последствия.

Например, для компьютеров Mac символьная константа `'\377'` будет преобразована в значение `-1`, поскольку ее значение будет рассматриваться как отрицательное для восьмибитового размера.

Вспомните, что в языке программирования C можно объявлять символьные переменные как тип `unsigned`, что позволяет избежать потенциальных проблем. Таким образом, символьная переменная типа `unsigned char` никогда не будет подвергаться знаковому расширению при приведении ее к целочисленному типу. Ее значение всегда будет больше или равно 0. Для восьмиразрядных символов переменные со знаком будут иметь значения от `-128` до `127` включительно. Беззнаковые символьные переменные будут иметь диапазон значений от 0 до `255` включительно.

Если вы хотите использовать знаковое расширение для символьных переменных, то вы должны объявлять переменные как тип `signed char`. В результате при приведении типов для символьных переменных будет производиться знаковое расширение даже на машинах, которые не делают этого по умолчанию.

Приведение аргументов

Прототипы объявлений используются для всех функций, которые представлены в этой книге. В главе 1, “Работа с функциями”, вы узнали, что использование прототипов объявлений позволяет размещать функцию и после того, как она будет вызвана в программе, и даже размещать необходимые функции в других файлах. Также было отмечено, что компилятор автоматически преобразовывает аргументы функций к тому типу, с которым они были объявлены. Но компилятор может это сделать только в том случае, если до обращения к функции он получил или объявление функции, или прототип объявления.

Вспомните, что если компилятор не зафиксировал ни объявления функции, ни прототипа объявления до обращения к функции, он предполагает, что функция возвращает значение типа `int`.

Компилятор также делает предположение о типах аргументов. При отсутствии информации о типах аргументов, компилятор автоматически преобразовывает типы аргументов `_Bool`, `char` или `short` в тип `int`, а аргументы типа `float` — в тип `double`.

Например, предположим, что компилятор встретил в программе следующие строки.

```
float x;  
...  
y = absoluteValue (x);
```

Не имея сведений о предварительном объявлении функции `absoluteValue` и не встретив ее прототипа объявления, компилятор генерирует код для преобразования аргумента `x` типа `float` в тип `double` и передает результат в функцию. Компилятор также предполагает, что функция возвращает значение типа `int`.

Но если функция `absoluteValue` где-то объявлена следующим образом

```
float absoluteValue (float x)
{
    if ( x < 0.0 )
        x = -x;
    return x;
}
```

то у вас возникнут трудности. Во-первых, функция возвращает значение типа `float`, даже если компилятор предполагает, что она возвращает значение типа `int`. Во-вторых, функция ожидает, что ей будет передан аргумент типа `float`, но на самом деле она получит значение типа `double`.

Поэтому хорошо запомните, что вы всегда должны включать прототипы объявлений для всех функций, которые используются в программе. Это защитит компилятор от ошибочных предположений насчет типов аргументов и типа возвращаемого значения.

Пока это все, что вы должны знать о типах данных. Далее вы ознакомитесь с тем, как работать с программами, которые можно разделить на несколько файлов. Глава 15, "Работа с большими программами", раскрывает все детали этой техники.

Упражнения

1. Задайте тип `FunctionPtr` (в помощь ключевого слова `typedef`), который будет представлять указатель на функцию, возвращающую значение типа `int` и не принимающую никаких аргументов. Обратитесь в главе 11, "Указатели", для уточнения того, как объявляются переменные такого типа.
2. Напишите функцию с именем `monthName`, которая принимает в качестве аргумента перечислимый тип `enum month` и возвращает указатель на строку символов, содержащую название месяца. Таким образом вы можете отобразить значение переменной типа `enum month` с помощью следующего утверждения.

```
printf ("%s\n", monthName (aMonth));
```

3. Допустим, что переменные объявлены следующим образом.

```
float f = 1.00;
short int i = 100;
long int l = 500L;
double d = 15.00;
```

Используя семь шагов по приведению типов операндов в выражениях, определите тип и значение результата для следующих выражений.

```
f + i
l / d
i / l + f
l * i
f / 2
i / (d + f)
l / (i * 2.0)
l + i / (double) l
```

Работа с большими программами

Все предыдущие программы, представленные в этой книге, были небольшими и относительно простыми. К сожалению, программы, которые вы будете писать на ассемблере для решения отдельных проблем, вероятнее всего, не будут такими маленькими и легкими. Изучению принципов работы с большими программами и посвящена эта глава. Как вы сможете убедиться, язык программирования С имеет все необходимые возможности для эффективной разработки больших программ. В дополнение к этому, вы можете использовать обслуживающие программы, о которых говорится в данной главе, с тем чтобы сделать работу с большими программами еще более удобной.

Разделение программы на несколько файлов

В отношении всех предыдущих программ этой книги предполагалось, что вся программа находится в одном файле, с которым можно работать с помощью текстового редактора, такого как `emacs`, `vim` или подобного им редактора для Windows, после чего осуществляется компиляция и реализация программы. В этот единственный файл включены все необходимые функции, но, разумеется, за исключением системных функций, таких как `printf` или `scanf`.

Стандартные заголовочные (подключаемые) файлы наподобие `stdio.h` или `stdlib.h` подключаются к вашей программе с помощью специальных утверждений. Такой подход вполне приемлем для небольших программ, т.е. таких программ, которые содержат не более 100 строк кода. Но при возрастании количества строк программы увеличивается и время на редактирование и повторную компиляцию программы.

К тому же зачастую для разработки одной большой программы привлекается несколько программистов. А вследствие работы нескольких программистов над одним файлом, к сожалению, процесс порой становится неуправляемым и создание программы представляется весьма проблематичным.

Язык программирования С поддерживает концепцию *модульного программирования*, при котором не требуется, чтобы все утверждения программы были включены в один файл. Это означает, что вы можете использовать для одного модуля единой программы один файл, для другого модуля — другой и т.д. Под модулем мы будем понимать или

отдельную функцию, или несколько взаимосвязанных функций, которые можно сгруппировать логически.

Если вы работаете в среде оконных инструментальных средств, таких как Metrowerks' CodeWarrior, Microsoft Visual Studio или Apple's Xcode, то работа с несколькими файлами для вас не будет представлять никакой сложности. Вы просто подключите нужный файл к проекту, с которым вы работаете, а все остальное среда программирования сделает за вас. Поэтому в следующем разделе мы рассмотрим работу с несколькими файлами. При этом мы предположим, что у вас нет интегрированной среды разработки. Таким образом, предполагается, что вы компилируете программу с помощью командной строки, непосредственно используя, например, команды `gcc` или `cc`.

Компиляция нескольких исходных файлов с помощью командной строки

Предположим, что вы концептуально разделили свою программу на три модуля и записали первый модуль в файл с именем `mod1.c`, программную часть второго модуля записали в файл `mod2.c` и основную часть программы с функцией `main` — в файл с именем `main.c`. Для того чтобы сообщить системе, что эти файлы действительно принадлежат одной и той же программе, вы просто включаете все три имени для этих файлов в команду, которую вы вводите с командной строки. Например, если вы используете компилятор `gcc`, то команда будет выглядеть следующим образом.

```
$ gcc mod1.c mod2.c main.c -o dbtest
```

И это будет иметь эффект *раздельной компиляции кода*, заключенного в модули `mod1.c`, `mod2.c` и `main.c`. Ошибки, обнаруженные в модулях `mod1.c`, `mod2.c` и `main.c`, будут выводиться компилятором отдельно для каждого модуля. Например, компилятор `gcc` может сделать вывод, подобный следующему.

```
mod2.c:10: mod2.c: In function 'foo':
mod2.c:10: error: 'i' undeclared (first use in this function)
mod2.c:10: error: (Each undeclared identifier is reported only once
mod2.c:10: error: for each function it appears in.)
```

Здесь компилятор указал, что в модуле `mod2.c` есть ошибка, которая находится в строке 10 в функции `foo`. Поскольку для модулей `mod1.c` и `main.c` не отображается никаких сообщений, то в этих модулях нет ошибок времени компиляции (Ошибки могут возникать и из-за заголовочных файлов, подключенных к модулю, т.е. необходимо корректировать заголовочный файл, а не модуль.)

Если в модуле находится ошибка, то разработчик обычно сразу редактирует модуль с целью устранить ошибку. В нашем случае, поскольку ошибка найдена только в модуле `mod2.c`, необходимо редактировать только этот модуль. Затем вы можете указать компилятору языка программирования C перекомпилировать модули для устранения ошибки.

```
$ gcc mod1.c mod2.c main.c -o dbtest
$
```

Поскольку в этом случае ошибок не найдено, исполняемый файл будет помещен в файл с именем `dbtest`. Компилятор обычно создает объектный код из модуля `mod.c` и сохраняет его в модуле `mod.o` по умолчанию. Большинство компиляторов для Windows работают аналогично, за исключением того, что помещают объектный модуль в файл с расширением `.obj` вместо расширения `.o`.

Обычно промежуточные объектные файлы автоматически разделяются после окончания компиляции. Некоторые компиляторы языка C (компилятор Unix C) сохраняют эти файлы вместе и при обработке нескольких модулей не разделяют их по окончании компиляции, и это можно использовать как преимущество при перекомпиляции программы после устранения ошибок в программе.

Например, в предыдущем примере модули `mod1.c` и `main.c` не имели ошибок времени компиляции, а значит, соответствующие объектные файлы `mod1.o` и `main.o`, можно не создавать вновь при перекомпиляции программы. Заменяя расширение `.c` в имени файла `mod.c` на расширение `.o` можно указать компилятору использовать уже существующие объектные модули вместо создания новых объектных модулей. Для этого необходимо использовать следующую команду для компиляции (используем компилятор `cc`), в результате чего будут использоваться существующие объектные модули.

```
$ cc mod1.o mod2.o main.o -o dbtest
```

Поскольку вы не делали изменений в модулях `mod1.c` и `main.c`, то нет необходимости их компилировать вновь, благодаря чему можно сэкономить время.

Если используемый вами компилятор автоматически стирает промежуточные файлы с расширением `.o`, вы и в этом случае можете использовать преимущество от наличия уже созданных объектных файлов, выполняя последовательную компиляцию, т.е. компилируя каждый модуль отдельно, используя опцию `-c` для команды. Эта опция указывает компилятору не компоновать ваш файл (что приводит к созданию только объектного модуля, а исполняемый модуль не создается) и оставлять созданные ранее промежуточные файлы. Поэтому, набрав команду

```
$ gcc -c mod2.c
```

вы скомпилируете исходный модуль `mod2.c` и получите объектный модуль `mod2.o`.

В общем, вы можете использовать последовательную компиляцию трех модулей из вашей программы `dbtest` с помощью следующей серии команд.

```
$ gcc -c mod1.c Compile mod1.c => mod1.o
$ gcc -c mod2.c Compile mod2.c => mod2.o
$ gcc -c main.c Compile main.c => main.o
$ gcc mod1.o mod2.o mod3.o -o dbtest Create executable
```

Все три исходных модуля будут скомпилированы отдельно. В нашем примере видно, что при этом не обнаружено никаких ошибок. Если же ошибки проявятся, то необходимо отредактировать соответствующий модуль и выполнить перекомпиляцию в последней строке.

```
$ gcc mod1.o mod2.o mod3.o
```

Как видим, перечислены только объектные модули и ни одного исходного модуля. В этом случае компилятор только скомпирует эти модули и сгенерирует исполняемый выходной файл с именем `dbtest`.

Если вы разобьете программу на большее количество исходных модулей, вы можете использовать тот же механизм раздельной компиляции и более эффективно работать с большой программой. Например, команды

```
$ gcc -c legal.c Compile legal.c, placing output in legal.o
$ gcc legal.o makemove.o exec.o enumerator.o evaluator.o display.o
-o superchess
```

можно использовать для компиляции программы, которая состоит из шести модулей и в которой только модуль `legal.c` должен быть перекомпилирован.

Из последнего раздела этой главы вы узнаете, что процесс раздельной компиляции можно автоматизировать с помощью инструментальной программы `make`. Автоматизированные среды разработки программ, о которых упоминалось выше в этой главе, постоянно отслеживают режим компиляции и перекомпилируют файлы только по мере необходимости.

Связь между модулями

Поскольку для отдельных модулей необходимо использовать методы из других модулей, то вполне понятна необходимость эффективного взаимодействия модулей из разных файлов. Если функции из одного файла необходимо вызвать функцию, содержащуюся в другом файле, то все можно сделать обычным образом. Как и для одного файла, в функцию передаются аргументы и используется возвращаемое значение. Разумеется, в файл, в котором находится вызывающая функция, необходимо включить прототип объявления функции, с тем чтобы компилятор знал типы передаваемых аргументов и тип возвращаемого значения. Как отмечалось в главе 14, “Еще о типах данных”, при отсутствии информации о функции компилятор автоматически при вызове функции присваивает возвращаемому значению тип `int` и приводит аргументы типа `short` или `char` к типу `int`, а аргументы типа `float` — к типу `double`.

Важно понимать, что если в команде задано более одного модуля, то компилятор компилирует каждый модуль независимо от других. Это означает, что никакой информацией о типах, содержащихся в других модулях, компилятор на момент компиляции одного модуля не обладает. Поэтому необходимо позаботиться о том, чтобы предоставить эту информацию о каждом модуле компилятору, с тем чтобы он мог правильно выполнить компиляцию.

Внешние переменные

Функции, содержащиеся в отдельных файлах, могут эффективно взаимодействовать с помощью *внешних переменных*, которые являются эффективным расширением глобальных переменных, обсуждаемых в главе 8, “Работа с функциями”.

Значения внешних переменных могут быть доступны и изменены из любого модуля. Внутри модуля, для которого необходим доступ к внешней переменной, эта переменная объявляется как обычно, за исключением того, что к ней при объявлении добавляется ключевое слово `extern`, размещаемое в начале объявления. Для системы это означает, что эта переменная может быть доступна из другого файла.

Предположим, что вы объявили переменную типа `int` с именем `moveNumber`, значение которой необходимо изменять из функций, расположенных в другом файле. Из главы 8 вы узнали, что если написать утверждение

```
int moveNumber = 0;
```

в начале программы, за пределами всех функций, то к этой переменной можно обращаться из всех функций программы. В этом случае переменная `moveNumber` считается глобальной переменной.

На самом деле такое объявление переменной `moveNumber` можно рассматривать не только как объявление глобальной переменной, но и как объявление внешней переменной. Для того чтобы эта переменная была доступна из других файлов, к предыдущему объявлению необходимо добавить ключевое слово `extern`, как показано ниже.

```
extern int moveNumber;
```

После такого объявления переменная `moveNumber` может быть доступна и ее значение может быть изменено из других модулей, в которых также используется такое объявление. То есть в других модулях должно быть сделано объявление переменной `moveNumber` с ключевым словом `extern`.

При работе с внешними переменными вы должны соблюдать одно очень важное правило. Переменная должна быть объявлена в определенном месте исходного файла. Это можно сделать двумя способами. Первый — это объявить переменную за пределами всех функций и не использовать ключевое слово `extern`.

```
int moveNumber;
```

Разумеется, переменной можно присвоить значение при объявлении, как об этом говорилось ранее.

Второй способ — это объявить внешнюю переменную внутри функции, помещая ключевое слово `extern` в начале объявления, и явно присвоить этой переменной значение, как показано ниже.

```
extern int moveNumber = 0;
```

Обратите внимание, что эти два способа являются в некотором роде взаимоисключающими. Когда вы работаете с внешними переменными, ключевое слово `extern` может быть пропущено в одном месте вашего исходного файла. И если вы ставите это ключевое слово, то вы обязательно должны присвоить переменной начальное значение.

Рассмотрим небольшую программу, в которой демонстрируется использование внешних переменных. Предположим, что в файле с именем `main.c` вы написали следующий код.

```
#include <stdio.h>
int i = 5;
int main (void)
{
    printf ("%i ", i);
    foo ();
    printf ("%i\n", i);
    return 0;
}
```

Объявление глобальной переменной `i` делает ее доступной в любом модуле, в котором используется соответствующее объявление этой переменной с ключевым словом `extern`. Предположим, вы написали следующие утверждения в файле с именем `foo.c`.

```
extern int i;
void foo (void)
{
    i = 100;
}
```

Совместно компилируя модули `main.c` и `foo.c` с помощью командной строки

```
$ gcc main.c foo.c
```

и затем выполняя программу, вы получите следующий вывод на терминал.

```
5 100
```

Такой вывод говорит о том, что функция `foo` способна обратиться к внешней переменной `i` и изменить ее значение.

Поскольку обращение к внешней переменной `i` производится только внутри функции `foo`, можно разместить объявление переменной `i` с ключевым словом `extern` внутри функции, как показано ниже.

```
void foo (void)
{
    extern int i;
    i = 100;
}
```

Но если обращение к внешней переменной `i` в файле `foo.c` производится из нескольких функций, то проще сделать объявление один раз в начале файла с ключевым словом `extern`. При этом необходимо учитывать тот факт, что в целях создания более защищенной программы лучше использовать объявления внешней переменной внутри функции, особенно если число таких функций невелико. Программа получится более организованной и к внешней переменной можно будет обратиться только отсюда, откуда это действительно необходимо.

При объявлении внешнего массива, нет необходимости задавать его размеры. Объявление наподобие

```
extern char text[];
```

позволит сослаться на символьный массив `text`, который объявлен где-то в другом месте. Но если внешний массив является многомерным, то размер последнего измерения все же должен быть объявлен. Таким образом, объявления

```
extern int matrix[][50];
```

достаточно для объявления двумерного массива с именем `matrix`, который содержит 50 колонок.

Статические объявления против внешних

Вы уже знаете, каждая переменная, объявленная за пределами всех функций, является не только глобальной, но и внешней переменной. Но возникает достаточно много ситуаций, когда необходимо объявить переменную только глобальной, но не внешней. Другими словами, возникает необходимость объявить глобальную переменную локально, только в одном модуле (файле). Это целесообразно в том случае, когда к этой переменной обращаются только те функции, которые расположены в данном файле, и больше никакие другие. Такое объявление можно сделать с помощью ключевого слова `static`. Утверждение

```
static int moveNumber = 0;
```

сделанное за пределами всех функций, позволит обращаться к переменной `moveNumber` в любом месте файла, в котором это объявление сделано, но не из функций, расположенных в других файлах.

Поэтому если вы хотите объявить глобальную переменную, к которой не должно быть доступа из других файлов, то объявляйте ее с ключевым словом `static`. Это делает программу более понятной и надежной. Статические переменные (объявленные с ключевым словом `static`) точно отражают смысл использования глобальной переменной и позволяют избежать конфликтов имен, который может возникнуть в том случае, если в другом модуле будет объявлена переменная с аналогичным именем.

Как отмечалось ранее, можно непосредственно вызывать функцию, объявленную в другом файле. В отличие от переменных, здесь нет необходимости специально указывать область видимости функции и можно не ставить ключевого слова `extern` в заголовке функции.

При объявлении функции она может быть описана как с ключевым словом `extern`, так и со словом `static`, но первое ключевое слово подставляется по умолчанию. Статические функции могут вызываться только из того файла, в котором они объявлены. Следовательно, если вы написали функцию с именем `squareRoot` и в начале заголовка функции поставили ключевое слово `static`, как показано ниже, то обратиться к этой функции можно только из того файла, в котором эта функция объявлена.

```
static double squareRoot (double x)
{
    ...
}
```

Такое объявление функции делает ее локальной в данном файле и обратиться к ней из другого файла уже будет невозможно. Здесь можно привести те же самые доводы, которые использовались при описании статических переменных.

На рис. 15.1 обобщены понятия связи между различными модулями. Здесь изображены два модуля: `mod1.c` и `mod2.c`. В модуле `mod1.c` объявлены две функции: `doSquare` и `main`. При этом возникают следующие связи: в функции `main` вызывается функция `doSquare`, которая в свою очередь вызывает функцию `square`. Эта функция объявлена в модуле `mod2.c`.

Поскольку функция `doSquare` объявлена как статическая, она может быть вызвана только из функций, расположенных в модуле `mod1.c`, и ни из каких других функций.

В модуле `mod1.c` объявлены две глобальные переменные: `x` и `result`, обе они именно тип `double`. Переменная `x` может быть доступна из любого модуля, который компилируется совместно с модулем `mod1.c`. С другой стороны, ключевое слово `static`, которое присутствует в объявлении переменной `result`, свидетельствует о том, что эта переменная может быть доступна только для функций, описанных в модуле `mod1.c` (`main` и `doSquare`).

Когда начинается выполнение программы, из функции `main` вызывается функция `doSquare`. Эта функция присваивает значение 2.0 глобальной переменной `x` и затем вызывает функцию `square`. Поскольку функция `square` находится в другом исходном файле (модуль `mod2.c`) и не возвращает тип `int`, функция `doSquare` включает соответствующее объявление в начале тела функции.

Функция `square` возвращает квадратный корень для значения глобальной переменной. Поскольку функции `square` необходим доступ к значению переменной, которая объявлена в другом файле (в `mod1.c`), то соответствующее объявление для переменной `x` с ключевым словом `extern` сделано в модуле `mod2.c` (в данном случае не имеет значения, где сделано объявление — в самой функции `square` или за ее пределами).

Значение, возвращаемой функцией `square`, присваивается глобальной переменной `result` внутри функции `doSquare`, которая вызывается в функции `main`. В функции `main` производится отображение глобальной переменной `result`. После выполнения этого примера, на экране терминала появляется значение 4.0 (квадратный корень числа 2.0).

Этот небольшой, хотя и не имеющий практического применения пример демонстрирует очень важное понятие взаимосвязи между модулями. Поэтому рекомендуем изучить его с должным вниманием, чтобы впоследствии эффективно работать с большими программами.

```

double x;
static double result;

static void doSquare (void)
{
    double square (void);

    x = 2.0;
    result = square ();
}

int main (void)
{
    doSquare ();
    printf ("%g\n", result);

    return 0;
}

```

mod1.c

```

extern double x;

double square(void)
{
    return x * x;
}

```

mod2.c

Рис. 15.1. Взаимосвязь между модулями

Эффективное использование заголовочных файлов

В главе 13, “Препроцессор”, была представлена концепция подключаемых файлов. Там было показано, как можно сгруппировать все часто используемые объявления внутри таких файлов и затем подключать их к программам, в которых необходимы эти объявления. Наибольшую пользу такие файлы приносят именно в больших программах, которые разделены на несколько программных модулей.

Если над программой работает несколько программистов, то подключаемые файлы используются как средство стандартизации при написании программы. Каждый программист будет использовать одни и те же объявления, которые имеют одинаковые значения.

Более того, каждый программист таким образом резервирует трудоемкие и требующие серьезной отладки задачи, задавая их объявления в каждом файле, где они будут использоваться в дальнейшем. На это надо обращать серьезное внимание при размещении объявлений общих структур, внешних переменных, конструкций typedef и объявлений прототипов функций в подключаемых файлах.

Различные модули больших программ будут всегда работать с общими структурами данными. Собирая объявления общих структур данных в одном модуле, вы исключите ошибки, связанные с тем, что различные модули будут содержать различные объявления для одних и тех же структур данных. Более того, если для какого-либо типа потребуется сделать изменения, то это изменение будет сделано только в одном месте — в подключаемом файле.

Вспомните структуру date из главы 9, “Работа со структурами”, оформленную как подключаемый файл, который можно использовать для работы со многими программными модулями. Эта структура также может послужить хорошим общим примером, в котором объединены многие концепции, которые вы изучили к настоящему времени.

```

// Заголовочный файл для работы с данными.
#include <stdbool.h>
// Перечислимые типы.

```

```

enum kMonth { January=1, February, March, April, May, June,
             July, August, September, October, November, December };
enum kDay { Sunday, Monday, Tuesday, Wednesday, Thursday, Friday };
struct date
{
    enum kMonth month;
    enum kDay day;
    int year;
};
// Типы данных.
typedef struct date Date;
// Функции для работы с данными.
Date dateUpdate (Date today);
int numberOfDays (Date d);
bool isLeapYear (Date d);
// Макроопределения для задания значений.
#define setDate(s,mm,dd,yy) s = (Date) {mm, dd, yy}
// Ссылки на внешние переменные.
extern Date todaysDate;

```

В заголовочном файле объявлены два перечислимых типа данных: kMonth и kDay, а также структура, в которой используются перечислимые типы данных. Ключевое слово typedef применяется для создания типа с именем Date, который используется для установки необходимых значений. Этот тип используется в объявлении функции, а также в объявлении внешней переменной с именем todaysDate, которую можно применить для задания текущей даты, причем и она должна быть объявлена в одном из исходных файлов.

Как пример использования заголовочных файлов, ниже приводится измененная версия функции dateUpdate из главы 9.

```

#include "date.h"
// Функция рассчитывает наступающую дату.
Date dateUpdate (Date today)
{
    Date tomorrow;
    if ( today.day != numberOfDays (today) )
        setDate (tomorrow, today.month, today.day + 1, today.year);
    else if ( today.month == December ) // Конец года.
        setDate (tomorrow, January, 1, today.year + 1);
    else // Конец месяца.
        setDate (tomorrow, today.month + 1, 1, today.year);
    return tomorrow;
}

```

Утилиты для работы с большими программами

Как уже упоминалось ранее, интегрированная среда разработки хорошо приспособлена для создания больших программ. Но если вы все еще продолжаете пользоваться командной строкой, то ниже для вас описываются несколько инструментальных средств, позволяющих облегчить разработку больших программ. Эти средства не

являются частью языка программирования C, но они рассчитаны на работу с языком C и вполне могут значительно сократить время разработки программ

Ниже приводятся несколько инструментальных средств, которые можно использовать при работе с большими программами. Если вы работаете с операционной системой Unix, то в ней вы в избытке найдете команды, которые также помогут при разработке программ. Но это только часть айсберга. Изучая язык сценариев, например для оболочки Unix, вы узнаете, что и его можно эффективно использовать для облегчения разработки программ с большим числом исходных модулей.

Утилита **make**

Эта мощная утилита (в версии GNU она называется `gnumake`) позволяет задать список файлов и их взаимосвязи в специальном установочном файле проекта (`Makefile`). Утилита `make` автоматически будет перекомпилировать файлы по мере необходимости. Причем вывод о необходимости компиляции будет делаться на основе времени модификации файла. Поэтому, если утилита `make` обнаружит, что исходный файл (расширение `.c`) по времени создан позднее, чем соответствующий объектный модуль (расширение `.o`), то она автоматически сгенерирует команды для перекомпиляции исходного файла и создаст новый объектный модуль. Можно даже учитывать исходные файлы, на которые есть ссылки в заголовочных файлах.

Например, вы можете указать модуль с именем `datefuncs.o` как зависимый от исходного файла `datefunc.c` и от заголовочного файла `date.h`. Теперь при каких-либо изменениях в заголовочном файле `date.h`, утилита `make` перекомпилирует файл `datefuncs.c`. Вывод будет сделан на основе того, что заголовочный файл по времени модифицирован позже, чем исходный файл.

Создадим простой установочный файл для использования с тремя модулями из этой главы. Предположим, что вы разместили этот файл в том же самом каталоге, где находятся исходные файлы.

```
$ cat Makefile
SRC = mod1.c mod2.c main.c
OBJ = mod1.o mod2.o main.o
PROG = dbtest
$(PROG) : $(OBJ)
gcc $(OBJ) -o $(PROG)
$(OBJ) : $(SRC)
```

Здесь приводится не детальное объяснение работы установочного файла, а только краткое пояснение. Во-первых, в нем задана последовательность исходных файлов (`SRC`), а также соответствующая последовательность объектных файлов (`OBJ`), имя исполняемой программы (`PROG`), и отдельные зависимости. Первая зависимость

```
$(PROG) : $(OBJ)
```

говорит о том, что исполняемый модуль зависит от объектных модулей. То есть, если один или более объектных модулей будут изменены, то исполняемый модуль необходимо скомпоновать вновь. То, как это сделать указано в следующей команде `gcc`, которая должна быть выполнена с помощью командной строки следующим образом:

```
gcc $(OBJ) -o $(PROG)
```

В последней строке установочного файла

```
$(OBJ) : $(SRC)
```

указывается на то, что объектный файл зависит от соответствующего исходного файла. Поэтому если исходный файл будет изменен, то его необходимо будет перекомпилировать и создать новый объектный файл. В утилите `make` для этого имеются встроенные правила. Вот что будет сделано при первом запуске утилиты `make`.

```
$ make
gcc -c -o mod1.o mod1.c
gcc -c -o mod2.o mod2.c
gcc -c -o main.o main.c
gcc mod1.o mod2.o main.o -o dbtest
$
```

Как видно, это очень удобно. Утилита `make` компилирует каждый исходный файл и затем компоует полученные объектные модули и создает исполняемый модуль. Но в случае ошибки времени компиляции, например, в модуле `mod2.c`, вывод, сделанный утилитой `make`, будет выглядеть следующим образом.

```
$ make
gcc -c -o mod1.o mod1.c
gcc -c -o mod2.o mod2.c
mod2.c: In function 'foo2':
mod2.c:3: error: 'i' undeclared (first use in this function)
mod2.c:3: error: (Each undeclared identifier is reported only once
mod2.c:3: error: for each function it appears in.)
make: *** [mod2.o] Error 1
$
```

Утилита `make` обнаружила ошибку при компиляции модуля `mod2.c` и прекратила дальнейшую обработку программы. Если вы исправите ошибку и опять запустите утилиту `make`, то получите следующее.

```
$ make
gcc -c -o mod2.o mod2.c
gcc -c -o main.o main.c
gcc mod1.o mod2.o main.o -o dbtest
$
```

Обратите внимание, что утилита `make` не выполняла повторной компиляции модуля `mod1.c`, т.к. она определила, что этот модуль уже корректно скомпилирован. В этом и заключается реальная эффективность и мощь утилиты `make`. Познакомившись с таким простым примером, вы уже можете использовать установочные файлы в своих программах. В приложении Д, “Ресурсы”, представлена более подробная информация об этой мощной утилите.

Утилита `cvs`

Это одна из многочисленных утилит для управления исходными файлами. Она производит автоматический контроль версий исходных файлов и отслеживает изменения, сделанные в модуле. Это позволяет при необходимости повторить отдельную версию программы (например, вернуться к более раннему коду, или создать отдельную версию для поддержки пользователей). С помощью утилиты `cvs` (базовая для Системы параллельных версий), вы обрабатываете программу применяя соответствующие опции, производите в ней изменения и затем повторно выполняете обработку программы (используя команду `cvs` с опцией `commit`).

Такой механизм позволяет избежать потенциальных конфликтов, которые могут возникнуть, если вносить изменения в исходный код будут несколько программистов. С помощью утилиты `cvs` программисты могут работать с исходным кодом, располагаясь в разных местах и получая доступ к коду через сеть.

Утилиты Unix: `ar`, `grep`, `sed` и т.д.

Наличие в системе Unix достаточного количества специальных команд позволяет разрабатывать большие программы под Unix более легко и эффективно. Например, вы можете создать пакет прикладных функций, которые планируете использовать часто или совместно. Аналогично тому, как вы связываете свою программу с подпрограммами из стандартных библиотек, используя опцию `-lm` при компоновке программы, вы можете задать и свою собственную библиотеку с помощью опции `-llib`.

На этапе компоновки, библиотека будет просмотрена с целью обнаружения функции, которая указана в программе. Все найденные функции будут извлечены из библиотек и объединены с вашей программой.

Другие команды, такие как `grep` и `sed` необходимы при поиске строк в файле или при изменениях в нескольких файлах. Например, с помощью команды `sed` вы легко можете найти и изменить имя отдельной переменной во всех исходных файлах, содержащих это имя. Команда `grep` просто ищет в одном или нескольких файлах заданную строку. Это поможет найти все вхождения переменной или функции в нескольких исходных файлах, или вхождения макроопределения в заголовочных файлах. Например, команда

```
$ grep todaysDate main.c
```

может использоваться для поиска всех строк, содержащих последовательность символов "todaysDate" в файле `main.c`. А команда

```
$ grep -n todaysDate *.c *.h
```

просмотрит все исходные и заголовочные файлы в текущем каталоге и отобразит все найденные вхождения в виде номера строки для соответствующего файла (такой режим задается с помощью опции `-n`).

Как вы могли убедиться, язык программирования C поддерживает разделение программы на небольшие модули и может производить независимую компиляцию этих модулей. Заголовочные файлы объединяют модули с помощью совместно используемых прототипов объявлений, макроопределений, объявлений структур, перечислений и т.д.

Если вы используете интегрированную среду разработки, то управление несколькими модулями программы будет проходить для вас почти незаметно.

Интегрированная среда разработки будет сама отслеживать файлы, которые необходимо перекомпилировать при внесении изменений в исходные файлы. Но если использовать компилятор командной строки, подобный `gcc`, то необходимо или самим производить все необходимые действия при внесении изменений, или несколько автоматизировать этот процесс с помощью утилиты `make`. Если вы работаете с командной строкой, то можете поискать и другие утилиты, которые помогут производить поиск в нескольких файлах одновременно, делать глобальные изменения и создавать библиотеки программ.

Ввод и вывод в языке С

Все символы, которые вы использовали до сих пор, вводились и выводились через окно терминала. Уточним еще раз, что под термином “терминал” в нашем случае понимается специально организованное окно Windows, в котором вы запускаете свои программы. В некоторых системах такое окно называют консолью. Используя терминал для ввода, т.е. желая ввести некоторую информацию, вы просто используете функции `scanf` или `getchar`. А для вывода данных, как известно, используется функция `printf`.

В стандарте языка программирования С нет специальных утверждений для выполнения операций ввода-вывода (I/O). Все операции ввода-вывода выполняются с помощью специально созданных для этих целей функций. Эти функции включены и в стандартные библиотеки языка программирования С.

Вспомните, как использовалось утверждение `include` в предыдущих программах. Оно применялось для того, чтобы подключить файл с необходимыми для ввода-вывода функциями.

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

Этот подключаемый файл содержит объявления и макроопределения, связанные с подпрограммами ввода-вывода из стандартных библиотек. Поэтому, когда у вас возникает необходимость использовать функции ввода-вывода из стандартных библиотек, вам необходимо подключить этот файл к своей программе.

В этой главе вы познакомитесь со многими функциями ввода-вывода из стандартных библиотек языка программирования С. К сожалению, из-за ограничений по объему невозможно провести полный анализ всех функций ввода-вывода. Для получения дополнительной информации обратитесь к приложению Б, “Стандартные библиотеки языка С”.

Ввод и вывод символов: getchar и putchar

Функция `getchar` весьма полезна в том случае, когда вам необходимо вывести один символ на экран терминала. Скоро вы узнаете, как можно разработать функцию с именем `readLine` для чтения целой строки с экрана терминала. Эта функция обращается к функции `getchar` до тех пор, пока не встретится и не будет считан символ новой строки.

Аналогичная функция для вывода одного символа на экран терминала называется `putchar`.

Вызов функции `putchar` очень простой. Единственный используемый аргумент — это тот символ, который необходимо отобразить. Поэтому вызов

```
putchar (c);
```

в котором аргумент `c` является переменной типа `char`, приведет к отображению на экране символа, заданного для переменной `c`. А вызов

```
putchar ('\n');
```

Приведет к выводу символа перехода на новую строку, в результате чего курсор переместится на начало следующей строки.

Форматированный ввод-вывод: printf и scanf

Функции `printf` и `scanf` использовались на протяжении всей книги. В этом разделе вы узнаете о всех возможных опциях для форматирования данных, используемых с этими функциями.

Первым параметром для обеих функций (`printf` и `scanf`) является указатель на символ. Он указывает на начало строки форматирования, которая определяет, как оставшиеся аргументы будут отображаться на экране в случае вызова функции `printf`, и как будут интерпретироваться считанные данные в случае вызова функции `scanf`.

Функция printf

Вы уже могли убедиться, изучая многие программы этой книги, что можно использовать символ `%` для преобразования других символов и получения большего контроля за выводом данных. Например, в листинге 5.3А было использовано число после символа преобразования для получения заданной точности вывода цифровых данных (была задана ширина поля). Написание символов форматирования в виде `%2i` приводит к тому, что отображаемое значение располагается в двух колонках и выравнивается по правому краю. Из упражнения 6 главы 5, “Программные циклы”, вы могли убедиться, что минус используется для выравнивания значения по левой границе поля.

Общий формат спецификации преобразования для функции `printf` представлен ниже.

```
%[flags][width][.prec][hlL]type
```

Необязательные поля заключены в квадратные скобки и при использовании должны располагаться в указанном порядке. В табл. 16.1, 16.2 и 16.3 приведены все возможные символы и значения, которые могут размещаться после символа `%` и до указания типа в строке форматирования.

Таблица 16.1. Флаги функции `printf`

Флаг	Описание
-	Выравнивание по левому краю
+	Значение предваряется знаком + или -
(space)	Ставится пробел для положительного значения
0	Заполнение нулем
#	Предшествует восьмеричному значению с 0, шестнадцатеричному значению с 0x (или 0X); отображение десятичной точки для типа <code>float</code> s; возможность оставить нули для форматов <code>g</code> или <code>G</code>

Таблица 16.2. Модификаторы точности и ширины поля для функции `printf`

Модификатор	Значение
number	Минимальный размер поля
*	Следующий аргумент как ширина поля
.number	Минимальное количество цифр для целого числа. Количество десятичных цифр для форматов <code>e</code> или <code>f</code> . Максимальное количество значащих цифр для формата <code>g</code> . Максимальное количество символов для формата <code>s</code>
.*	Следующий аргумент как значение точности

Таблица 16.3. Модификаторы типа для функции `printf`

Тип	Значение
hh	Отображает целое число как символ
h*	Отображает тип <code>short integer</code>
l*	Отображает тип <code>long integer</code>
ll*	Отображает тип <code>long long integer</code>
L	Отображает тип <code>long double</code>
j*	Отображает значение <code>intmax_t</code> или <code>uintmax_t</code>
t*	Отображает значение <code>ptrdiff_t</code>
z*	Отображает значение <code>size_t</code>

Замечание. Эти модификаторы также могут размещаться перед символом преобразования `p` для указания того, что соответствующий указатель является спецификатором типа. В табл. 16.4 перечислены все символы преобразования, которые могут быть заданы в строке форматирования.

Таблица 16.4. Символы преобразования для функции `printf`

Символ	Отображает
i или d	Целое число
u	Целое число без знака
o	Восьмеричное целое число
x	Шестнадцатеричное целое число, используются символы a – f
X	Шестнадцатеричное целое число, используются символы A – F
f или F	Вещественное число, шесть цифр по умолчанию
e или E	Вещественное число в экспоненциальном формате (вывод в нижнем регистре при e, и в верхнем при E)
g	Вещественное число в f- или e-формате
G	Вещественное число в F- или E-формате
a или A	Вещественное число в шестнадцатеричном формате 0xd.ddddp±d
c	Один символ
s	Строка с нулевым окончанием
p	Указатель
n	Ничего не печатать, сохранять до тех пор, пока не будут вызваны соответствующим аргументом (см. примеч. в табл. 16.3)
%	Знак процента

Может показаться, что табл. 16.1 – 16.4 довольно большие и содержат избыточные строки. Но, как вы убедитесь, большинство различных комбинаций можно эффективно использовать для задания необходимого формата вывода. Для лучшего понимания вопроса о форматах вывода необходимо реально попробовать сделать вывод для различных ситуаций. При этом надо учитывать следующее: количество аргументов при вызове функции `printf` должно соответствовать количеству символов % в строке форматирования (за исключением %%). При использовании символа * в поле для ширины или модификатора точности также необходим аргумент для каждого символа *.

В программе из листинга 16.1 приведены некоторые из возможных комбинаций использования форматирования для функции `printf`.

Листинг 16.1. Демонстрация форматирования для функции `printf`

```
// В программе показаны различные способы форматирования вывода.
#include <stdio.h>
int main (void)
{
    char c = 'X';
    char s[] = "abcdefghijklmnoprstuvwxyz";
    int i = 425;
    short int j = 17;
    unsigned int u = 0xf179U;
    long int l = 75000L;
    long long int L = 0x1234567812345678LL;
    float f = 12.978F;
    double d = -97.4583;
```

```

char *cp = &c;
int *ip = &i;
int c1, c2;

printf ("Integers:\n");
printf ("%i %o %x %u\n", i, i, i, i);
printf ("%x %X %#x %#X\n", i, i, i, i);
printf ("%i %i %i %07i %.7i\n", i, i, i, i);
printf ("%i %o %x %u\n", j, j, j, j);
printf ("%i %o %x %u\n", u, u, u, u);
printf ("%ld %lo %lx %lu\n", l, l, l, l);
printf ("%lli %llo %llx %llu\n", L, L, L, L);

printf ("\nFloats and Doubles:\n");
printf ("%f %e %g\n", f, f, f);
printf ("%2f %2e\n", f, f);
printf ("%0f %0e\n", f, f);
printf ("%7.2f %7.2e\n", f, f);
printf ("%f %e %g\n", d, d, d);
printf ("%.*f\n", 3, d);
printf ("%*.f\n", 8, 2, d);

printf ("\nCharacters:\n");
printf ("%c\n", c);
printf ("%3c%3c\n", c, c);
printf ("%x\n", c);

printf ("\nStrings:\n");
printf ("%s\n", s);
printf ("%5s\n", s);
printf ("%30s\n", s);
printf ("%20.5s\n", s);
printf ("% -20.5s\n", s);

printf ("\nPointers:\n");
printf ("%p %p\n", ip, cp);
printf ("This%n is fun.%n\n", &c1, &c2);
printf ("c1 = %i, c2 = %i\n", c1, c2);

return 0;
}

```

Листинг 16.1. Вывод

```

Integers:
425 651 1a9 425
1a9 1A9 0x1a9 0X1A9
+425 425 0000425 0000425
17 21 11 17
61817 170571 f179 61817
75000 222370 124f8 75000
1311768465173141112 110642547402215053170 1234567812345678
1311768465173141112
Floats and Doubles:
12.978000 1.297800e+01 12.978

```



```
12.98 1.30e+01
13 1e+01
12.98 1.30e+01
-97.458300 -9.745830e+01 -97.4583
-97.458
-97.46
Characters:
X
X X
58
Strings:
abcdefghijklmnopqrstuvwxyz
abcde
abcdefghijklmnopqrstuvwxyz
abcde
abcde
abcde
Pointers:
0xbffffc20 0xbffffbf0
This is fun.
c1 = 4, c2 = 12
```

Очень важно в полной мере понимать процесс форматирования вывода, и для этого стоит потратить некоторое время и прочитать подробное объяснение. В первых строках выводятся целые числа следующих типов: short, long, unsigned, а также обычные int. Отображается переменная *i* в десятичном (%i), восьмеричном (%o), шестнадцатеричном (%x) и беззнаковом (%u) форматах. Обратите внимание, что восьмеричные числа не начинаются с цифры 0 при отображении.

В следующей строке вывода снова отображается переменная *i*. Сначала переменная *i* отображается в шестнадцатеричном формате с помощью символов форматирования %x. Использование заглавной буквы X (%#X) приводит к тому, что при выводе будут использоваться символы верхнего регистра A–F вместо символов нижнего регистра, которые обычно используются при выводе шестнадцатеричных значений. Модификатор # (%#x) способствует появлению префикса 0x перед выводимым числом. Это происходит в том случае, когда символ X используется как символ преобразования.

При четвертом вызове функции printf сначала используется флаг +, который свидетельствует о необходимости ставить знак + при выводе значения, даже если значение положительное (по умолчанию знак + не отображается). Затем используется модификатор “пробел” для получения пробела перед выводом положительного значения. (Иногда это используется для выравнивания данных, которые могут быть как положительными, так и отрицательными. При этом положительные значения сдвигаются на одну позицию, которую у отрицательных значений занимает знак минуса). Затем используется комбинация форматирования %07 для отображения значения переменной *i*, выровненного по правому краю в поле шириной в семь символов. Флаг 0 способствует заполнению свободных мест нулями. Именно поэтому перед числом 425 находятся четыре нуля. Окончательное преобразование в этом вызове формируется последовательностью символов %.7i, которая используется для отображения значения переменной *i* с помощью не более 7 цифр. Эффект будет такой же, как и в случае символов форматирования %07i: отобразятся четыре нуля с последующими цифрами 425.

При пятом вызове функции printf отображается значение переменной *j* типа short int в различных форматах.

Любой формат для отображения целых чисел может использоваться для отображения типа `short int`. В следующем вызове функции `printf` показано, что произойдет, когда формат `%i` будет использован для отображения значения типа `unsigned int`. Поскольку значение, присвоенное переменной `u`, больше, чем максимально возможное число, которое может быть сохранено в переменной типа `signed int` для той системы, на которой эта программа запущена, то при использовании формата `%i` оно отображается как отрицательное число.

Затем очередной вызов функции `printf` из этого ряда показывает использование модификатора `l` для отображения целых чисел типа `long`, после чего отображаются значения типа `long long`.

Следующая последовательность вызовов демонстрирует возможность форматирования для отображения вещественных чисел форматов `float` и `double`. В первой строке из этой последовательности показан результат отображения значения типа `float` с помощью форматов `%f`, `%e` и `%g`. Если не оговорено другое, форматы `%f` и `%e` отображают шесть десятичных цифр. Если используется формат `%g`, то решение об окончательном отображении в формате `%f` или `%e` принимается в самой функции, в зависимости от размера значения и необходимой точности. Если экспонента меньше чем `-4` или больше чем дополнительно заданная точность (по умолчанию `6`), то используется формат `%e`, в противном случае используется формат `%f`. В любом случае, последующие нули автоматически удаляются, а десятичная точка отображается только в том случае, если отображаются значимые цифры. Вообще, формат `%g` считается наилучшим форматом для отображения вещественных чисел.

В следующей строке используется модификатор точности `.2`, который ограничивает количество отображаемых десятичных значений для переменной `f` двумя разрядами. Как видим, функция `printf` округляет значение переменной `f`. В следующей строке показано использование модификатора точности `.0`, когда подавляется вывод десятичных значений даже при символах форматирования `%f`. Но значение переменной при этом тоже округляется.

В следующем вызове функции `printf` в целях генерации вывода используется модификатор `7.2`, свидетельствующий о том, что для вывода нужно использовать как минимум семь колонок, из которых две определяют дробную часть (точность). Поскольку для вывода не всегда используются все семь колонок, то производится выравнивание по правой границе, а оставшиеся места заполняются пробелами.

В следующих трех строках вывода в различных форматах отображается переменная `d` типа `double`. Аналогичные форматы используются для отображения значений типов `float`, поскольку тип `float` автоматически приводится в тип `double` при передаче в функцию аргументов, о чем говорилось ранее. Вызов функции

```
printf ("%.*f\n", 3, d);
```

приводит к тому, что значение переменной `d` будет отображено в виде трех десятичных цифр. Звездочка после десятичной точки в строке форматирования свидетельствует о том, что функция `printf` должна принять следующий аргумент как значение точности. В нашем случае этим аргументом будет число `3`. Этот аргумент может быть выражен и с помощью переменной, например, следующим образом

```
printf ("%.*f\n", accuracy, d);
```

что удобно при динамическом изменении формата вывода.

В последней строке для вывода значений типа `float` и `double`, показано использование символов форматирования `%. *f` для отображения значения переменной `d`. В этом случае оба поля для ширины и точности заполняются с помощью аргументов,

передаваемых в функцию, о чем свидетельствуют две звездочки в строке форматирования. Поскольку первым аргументом после строки форматирования будет число 8, то оно принимается как ширина поля. Вторым аргументом 2 принимается как точность. Поэтому значение переменной `d` отображается с точностью 2 на поле шириной восемь цифр. Обратите внимание, что знак минуса, как и знак десятичной точки, включен в ширину поля. Это справедливо для любых спецификаций поля.

В следующей группе вывода используется символьная переменная `s`, которой изначально присвоено значение `X` и которая отображается в различных форматах. Сначала она отображается с помощью стандартного для символов формата `%s`. Затем она отображается дважды с шириной поля 3. В результате символ отображается с двумя предшествующими пробелами.

Символ можно отобразить с помощью любого формата, используемого для отображения целых чисел. В очередной строке значение для символьной переменной `s` отображается в шестнадцатеричном виде. Полученный вывод свидетельствует о том, что на используемой машине символ `X` имеет внутреннее представление как шестнадцатеричное число 58.

В окончательной серии выводов отображается строковая константа `s`. Сначала она отображается в принятом для строк формате `%s`, а затем с модификатором точности 5, и при этом должны отобразиться только первые пять символов строки, что приводит к отображению пяти первых букв алфавита.

В третьей строке этой серии строка отображается вновь, но с использованием модификатора ширины поля, равной 30. Как можно видеть, строка отображается выровненной по правой границе поля размером 30 символов.

В последних двух строках этой серии в поле шириной 20 символов отображаются пять символов строки `s`. Сначала эти пять символов отображаются выровненными по правой стороне поля. Затем в символах форматирования используется знак минус, с тем чтобы выравнивание происходило по левой стороне поля. В данном случае используется формат `%-20.5s`, при котором сначала выводится 5 символов, за которыми следуют 15 пробелов.

Формат `%p` используется для отображения значений указателей. В данном случае отображается указатель на целое число `ip` и указатель на символ `cp`. Необходимо отметить, что полученные вами результаты, вероятнее всего, будут отличаться от приведенных в книге, поскольку адреса будут другими.

Вывод при использовании формата `%p` будет зависеть от реализации, и в нашем случае указатель отображаются в шестнадцатеричном виде. В соответствии с выводом, указатель на переменную содержит шестнадцатеричный адрес `bffffc20`, а указатель на переменную `cp` содержит адрес `bffffbf0`.

В окончательной серии показано использование формата `%n`. В нашем случае соответствующий аргумент функции `printf` должен иметь тип указателя на целое число при типе модификатора `hh`, `h`, `l`, `ll`, `j`, `z` или `t`. Функция `printf` сохраняет количество символов, которые были выведены, как целое число для соответствующего аргумента. Поэтому в первом случае формат `%n` приведет к тому, что функция `printf` сохранит значение 4 для целочисленной переменной `c1`, поскольку именно это число символов было записано перед форматом `%n`. Второе использование формата `%n` приведет к присвоению переменной `c2` значения 12, т.к. именно такое количество символов было отображено до этого. Обратите внимание, что использование символов `%n` внутри строки форматирования никак не влияет на вывод, сделанный функцией `printf`.

Функция scanf

Подобно функции `printf`, довольно много форматов, которые уже были описаны, могут быть использованы в строке форматирования для функции `scanf`. Как и в случае с функцией `printf`, функция `scanf` использует дополнительные модификаторы между символом `%` и соответствующим символом. Все эти модификаторы приведены в табл./ 16.5. Возможные преобразования символов приведены в табл. 16.6.

Когда функция `scanf` проверяет входной поток для считывания значения, она всегда пропускает все “неотображаемые” символы, к которым относятся символ пробела, символ горизонтальной табуляции (`\t`), вертикальной табуляции (`\v`), перевод каретки (`\r`), новой строки (`\n`) или переход к следующей странице (`\f`). Исключением является формат `%c` — в этом случае очередной символ потока, независимо от того, что он из себя представляет, считывается и учитывается при выводе.

Таблица 16.5. Модификаторы преобразования для функции `scanf`

Модификатор	Значение
*	Поле пропускается и не присваивается
size	Максимальный размер входного поля
hh	Значение сохраняется как символ со знаком или без знака
h	Значение сохраняется как тип <code>short int</code>
l	Значение сохраняется как тип <code>long int</code> , <code>double</code> или <code>wchar_t</code>
j, z или t	Значение сохраняется как тип <code>size_t</code> (<code>%j</code>), <code>ptrdiff_t</code> (<code>%z</code>), <code>intmax_t</code> или <code>uintmax_t</code> (<code>%t</code>)
ll	Значение сохраняется как тип <code>long long int</code>
L	Значение сохраняется как тип <code>long double</code>
type	Символ преобразования

Таблица 16.6. Преобразование символов для функции `scanf`

Символ	Действие
d	Значение считывается как десятичное число, соответствующий аргумент является указателем на тип <code>int</code> , но если используются модификаторы <code>h</code> , <code>l</code> или <code>ll</code> , то аргумент является указателем на <code>short</code> , <code>long</code> или <code>long long int</code> соответственно
i	Аналогично <code>%d</code> , за исключением того, что восьмеричные числа (префикс <code>0</code>) и шестнадцатеричные числа (префикс <code>0x</code> или <code>0X</code>) также могут считываться
U	Значение считывается как целое число, и соответствующий аргумент является указателем на целое число без знака
o	Значение считывается как восьмеричное число, которое дополнительно может быть с префиксом <code>0</code> . Соответствующий аргумент является указателем на тип <code>int</code> , за исключением тех случаев, когда используются модификаторы <code>h</code> , <code>l</code> или <code>ll</code> с предшествующим символом <code>o</code> . В этом случае аргумент указывает на тип <code>short</code> , <code>long</code> или <code>long long int</code> соответственно.

Символ	Действие
x	Значение считывается как шестнадцатеричное число, которое может быть с дополнительным префиксом 0x или 0X. Соответствующий аргумент является указателем на тип <code>unsigned int</code> , за исключением использования модификаторов <code>h</code> , <code>l</code> или <code>ll</code>
a, e, f или g	Значение считывается как вещественное число, которое дополнительно может быть со знаком <code>+</code> или <code>-</code> , а также может быть выражено в экспоненциальном виде (как <code>3.45e-3</code>). Соответствующий аргумент указывает на тип <code>float</code> , за исключением случаев использования модификаторов <code>l</code> или <code>L</code> , когда аргумент будет указывать на тип <code>double</code> или <code>long double</code> соответственно
c	Значение считывается как один символ. Считывается любой символ, даже если это пробел, символ табуляции или новой строки, а также символ новой страницы. Соответствующий аргумент является указателем на тип <code>char</code> . Дополнительно перед символом <code>c</code> можно указать число считываемых символов
s	Значение считывается как последовательность символов. Последовательность начинается с первого отображаемого символа и заканчивается первым неотображаемым символом. Соответствующий аргумент является указателем на массив символов, который должен иметь достаточный размер для включения всех считанных символов плюс символ конца строки, который добавляется автоматически в конец строки. Если символу <code>s</code> предшествует число, то считывается указанное число символов. Если до этого не встретится неотображаемый символ
[...]	Символы, заключенные в квадратные скобки, свидетельствуют о том, что должна быть считана символьная строка, аналогично формату <code>%s</code> . Символы в скобках относятся к тем символам, которые могут быть считаны. Если встретится какой-либо другой символ, ввод строки прекращается. Эти символы могут иметь “инверсный” смысл, т.е. являться теми символами, которые нельзя считывать, для чего перед ними ставится знак <code>^</code> . В этом случае при считывании такого символа ввод прекращается
n	Ничего не считывать. Количество пропущенных таким образом символов записывается в соответствующий аргумент
p	Значение считывается как указатель на выражение для формата, аналогичного формату преобразования символов <code>%p</code> для функции <code>printf</code> . Соответствующий аргумент является указателем на тип <code>void</code>
%	Следующим отображаемым символом во входном потоке должен быть символ <code>%</code>

Когда функция `scanf` считывает отдельное значение, считывание прекращается сразу, как только считано количество символов, составляющих ширину поля (если задано), или до тех пор, пока не встретится запрещенный для данного случая символ. В случае целых чисел разрешена только последовательность цифр со знаком, которые

могут быть расширены буквами a–f или A–F в случае шестнадцатеричного значения (десятичное: 0–9, восьмеричное: 0–7, шестнадцатеричное: 0–9, a–f или A–F).

Для типа `float` допустимыми символами являются десятичные цифры со знаком с последующей десятичной точкой, за которой также идут десятичные цифры. За этими цифрами может идти буква `e` или `E` и необязательный знак экспоненты. В случае формата `%a`, шестнадцатеричное вещественное число может быть записано в предшествующими символами `0x`, за которыми следует последовательность шестнадцатеричных цифр с необязательной десятичной точкой и необязательной экспонентой с буквой `p` или `P`.

Для символьных строк, которые считываются с помощью формата `%s`, любой отображаемый символ является допустимым. При считывании строки с помощью формата `%c`, допустимыми являются все символы. Наконец, в случае использования строки в квадратных скобках, допустимыми являются те символы, которые записаны в квадратных скобках (которые могут быть и недопустимыми, если перед ними стоит символ `^`).

Вспомните, что вы узнали в главе 9, “Структуры”, когда изучали программу, в которой производился запрос к пользователю на введение времени с терминала. Все специальные символы, которые встречались в строке форматирования при вызове функции `scanf`, использовались при вводе. Например, вызов функции `scanf`

```
scanf ("%i:%i:%i", &hour, &minutes, &seconds);
```

означал, что должны быть считаны три значения и сохранены в переменных `hour`, `minutes` и `seconds` соответственно. Внутри строки форматирования используется символ `:`, который служит для разделения вводимых значений.

Если ожидается, что при вводе будет использоваться знак процента, то в строке форматирования должен стоять удвоенный знак процента (`%%`).

```
scanf ("%i%%", &percentage);
```

Неотображаемые символы внутри строки форматирования соответствуют произвольному количеству неотображаемых символов в строке ввода. Поэтому вызов

```
scanf ("%i%c", &i, &c);
```

для текстовой строки “29 w” приведет к тому, что переменной `i` будет присвоено значение 29, а переменной `c` будет присвоен символ пробела, который следует непосредственно за числом 29. Для того чтобы выполнить корректный ввод, необходимо написать следующее.

```
scanf ("%i %c", &i, &c);
```

И для той же самой текстовой строки будет сделан правильный ввод, т.е. переменной `i` будет присвоено значение 29, а переменной `c` будет присвоен символ `'w'`, поскольку символ пробела в строке форматирования приводит к тому, что функция `scanf` будет игнорировать все неотображаемые символы после числа 29, пока не встретится допустимый символ.

В табл. 16.5 отмечено, что звездочку можно использовать для пропуска полей. Если сделать вызов

```
scanf ("%i %5c %*f %s", &i1, text, string);
```

и набрать для ввода следующую текстовую строку

```
144abcde 736.55 (wine and cheese)
```

то значение 144 будет сохранено в переменной `i1`, пять символов "abcde" будут сохранены в символьном массиве `text`, следующее вещественное значение 736.55 будет пропущено и наконец текстовая строка "(wine" будет сохранена в переменной `string` и завершена символом `null`. Следующий вызов функции `scanf` продолжит ввод с того места, где закончился предыдущий ввод. Следовательно, если сделать такой вызов функции

```
scanf ("%s %s %i", string2, string3, &i2);
```

то будут сохранены строка "and" в переменной `string2` и строка "cheese)" в переменной `string3`, после чего функция будет ожидать ввода целочисленного значения.

Помните, что функция `scanf` использует указатели на переменные, в которые должно помещаться введенное значение. Из главы 11 "Указатели", вы узнали, почему следует поступать именно так. Функция `scanf` должна изменить значение этих переменных на те значения, которые были считаны, а сделать это можно только с помощью указателей.

Также помните, что в целях создания указателя на массив, необходимо записать только имя массива. Поэтому переменная `text` является указателем на массив символов достаточного размера, чтобы вместить строку при следующем вызове функции `scanf`.

```
scanf ("%80c", text);
```

После этого вызова в переменной `text` будет сохранено 80 символов из входного потока. Вызов функции

```
scanf ("%[^/]", text);
```

свидетельствует о том, что будет считана строка из любых символов за исключением обратной черты (`\`). Используя эту функцию для ввода следующей строки

```
(wine and cheese)/
```

вы сохраните в переменной `text` только строку "(wine and cheese)", поскольку символ "`\`" в данном случае не является допустимым, но он может быть учтен при следующем вызове функции `scanf`.

Для считывания целой строки в символьный массив `buf` необходимо задать, чтобы символ перехода на новую строку был символом окончания строки.

```
scanf ("%[^\n]\n", buf);
```

Символ новой строки находится за пределами квадратных скобок и таким образом что указывает функции `scanf` на то, что этот символ не надо считывать при последующем вызове функции `scanf`. Вспомните, что функция `scanf` всегда продолжает считывание с того места, где она остановилась в предыдущий раз.

Когда значение, подготовленное для ввода, не соответствует тому, что ожидает функция `scanf` (например, набран символ `X`, когда ожидается целочисленное значение), то функция прекращает дальнейшее считывание и происходит выход из функции. Поскольку функция возвращает количество успешно считанных значений, которые присваиваются переменным вашей программы, то это количество необходимо проверить и убедиться, что при вводе не сделано никаких ошибок. Например, вызов функции

```
if ( scanf ("%i %f %i", &i, &f, &l) != 3 )  
    printf ("Error on input\n");
```

производит проверку, что функция `scanf` успешно ввела и присвоила три значения. Если это не так, то отображается соответствующее сообщение.

Помните, что возвращаемое функцией `scanf` число соответствует количеству успешно считанных и присвоенных значений, поэтому вызов:

```
scanf ("%i %d %i", &i1, &i3)
```

возвратит число 2, а не 3, т.к. было считано и присвоено только два целых числа, а одно между ними было пропущено. Обратите внимание, что символы `%n`, использованные для получения количества считанных символов, не включаются в число возвращаемых функцией `scanf` значений.

Поэкспериментируйте с различными форматами ввода для функции `scanf`. Как и в случае с функцией `printf`, должное понимание различных форматов ввода будет достигнуто только после достаточной практики реального программирования.

Операции ввода-вывода для файлов

До сих пор, когда вы вызывали функцию `scanf` в любой из программ, то данные, которые необходимы для обработки ввода, извлекались из текста, вводимого через терминал. Аналогично, все вызовы функции `printf` приводили к отображению выводимых данных в окне терминала. В этом разделе вы узнаете, как вводить и выводить данные не в окне терминала, а использовать для этих целей файл.

Перенаправление ввода-вывода в файл

Как чтение, так и запись в файл может очень легко выполняться в различных операционных системах (Unix или Windows) без дополнительных изменений в программе. Например, если вы хотите записывать результаты работы программы в файл с именем `data`, то все, что необходимо сделать в системах Unix или Windows, — это перенаправить вывод от окна терминала в файл с именем `data` без изменений в программе с помощью следующей команды.

```
prog > data
```

Эта команда указывает системе выполнить программу `prog`, но сделать вывод не в окно терминала, а записать выводимые данные в файл с именем `data`. Поэтому все значения, выводимые функцией `printf`, не будут отображаться в окне терминала, а будут перенаправлены и записаны в файл с именем `data`.

Для лучшего понимания того, как это работает, возьмите самую первую программу из листинга 3.1 и скомпилируйте ее обычным способом. Затем выполните программу, набрав ее имя, например, `prog1`.

```
prog1
```

После этого вы получите на экране терминала следующую фразу.

Программирование забавно.

Теперь наберите следующую команду

```
prog1 > data
```

и обратите внимание, что на экране терминала вы ничего не увидите. Это произошло потому, что вывод, сделанный программой, был перенаправлен в файл с именем

data. Если после этого вы проверите содержимое файла data, то вы увидите там следующую строку.

Программирование забавно.

Это должно убедить вас в том, что вывод, сделанный вашей программой, был отправлен в файл таким образом, как это описано выше. Вы можете повторить предыдущую последовательность команд с программой, которая делает более интенсивный вывод данных, занимая несколько строк текста, и убедиться, что и в этом случае все будет работать именно так, т.е. весь вывод будет выполняться в файл.

Вы можете делать подобные перенаправления для всех выводов вашей программы. При вызове любой функции, которая обычно считывает данные из окна терминала, такой как `scanf` и `getchar`, можно легко заставить программу считывать данные не с терминала, а из файла. Программа из листинга 5.8 производит реверс (изменение порядка следования) цифр числа. В программе используется функция `scanf` для считывания числа, которое необходимо реверсировать, с окна терминала. Но вы можете легко заставить программу считывать значения не с окна терминала, а с файла с именем `number`, просто перенаправляя ввод программы во время выполнения. Если программа называется `reverse`, то следующая команда, введенная в командной строке, позволит вам легко это сделать.

```
reverse < number
```

Если в файле с именем `number` будет находиться число 2001 до того, как будет введена предыдущая команда, то после выполнения программы в окне терминала появится следующее.

```
Enter your number.  
1002
```

Обратите внимание, что запрос программы на ввод числа был сделан в окне терминала, но при этом программа не ждет ввода значения с терминала. Это происходит потому, что ввод программы, но не вывод, перенаправлен на файл с именем `number`. Поэтому данные для функции `scanf` будут считаны из файла с именем `number`, а не с окна терминала. При этом информация будет передана в программу так, как будто она была считана с окна терминала. То есть для функции `scanf` абсолютно безразлично, откуда ей были предоставлены данные, из файла или окна терминала, функция “беспокоится” только о том, чтобы эти данные были правильно сформатированы.

Вы вполне можете одновременно перенаправить как ввод, так и вывод. Такая команда, как

```
reverse < number > data
```

заставит программу наподобие `reverse` считывать все предназначенные для нее данные из файла `number` и записывать все выводимые данные в файл с именем `data`. Таким образом, если вы выполните такую команду для программы из листинга 5.8, то все входные данные вновь будут взяты из файла `number`, а запись выводимых данных будет производиться в файл `data`.

Такой метод перенаправления ввода или вывода для программы используется довольно часто. Например, вы пишете статью для журнала и ее окончательный вариант хотите сохранить в файле с именем `article`. С помощью программы из листинга 10.8 можно подсчитать количество слов, введенных с терминала, для чего необходимо использовать следующую команду.

```
wordcount < article
```

В системе Unix есть стандартная программа `wc`, которая также позволяет подсчитывать количество слов, но при этом имейте в виду, что все эти программы работают только с текстовыми файлами, а не с файлами типа `*.doc`.

Вы также не должны забывать вставлять символ возврата каретки в конце записи в файл `article`, поскольку программа может распознать конец данных в файле только по наличию одиночного символа новой строки.

Обратите внимание, что описанные выше перенаправления не являются частью языка программирования C. Это означает, что есть операционные системы, в которых такое перенаправление не работает, но, к счастью, не во всех.

Конец файла

Предыдущее упоминание о конце данных в файле заслуживает более подробного рассмотрения. При работе с файлами такое состояние называется *конец файла* (end of file). Состояние *конец файла* возникает тогда, когда считан последний фрагмент данных из файла. Попытка считать данные из файла после возникновения этого состояния приведет к аварийному завершению программы или даже к бесконечному программному циклу. К счастью, большинство функций из стандартной библиотеки ввода-вывода возвращают специальный флаг, сигнализирующий о том, что достигнут конец файла. Значение этого флага имеет специальное обозначение, которое выражается в виде аббревиатуры EOF (End Of File) и которое определено в стандартном заголовочном файле `stdio.h`.

Как пример использования символа конца файла EOF, мы используем программу из листинга 16.2, в которой для считывания данных используется функция `getchar`. Считанные данные отображаются в окне терминала до тех пор, пока не встретится конец файла. Обратите внимание на выражение, включенное в цикл `while`. Как можно видеть, присваивание не выделено в отдельное утверждение.

Листинг 16.2. Копирование символов из стандартного входа в стандартный выход

```
// Листинг to echo characters until an end of file
#include <stdio.h>
int main (void)
{
    int c;

    while ( (c = getchar ()) != EOF )
        putchar (c);

    return 0;
}
```

После компиляции при выполнении программы необходимо сделать перенаправление ввода из файла с помощью следующей команды:

```
copyprog < infile
```

После этого программа отобразит содержимое файла `infile` в окне терминала. На самом деле программа выполняет такие базовые функции, как, например, команда `cat` в операционной системе Unix, которую можно использовать для отображения содержимого любого файла на экране терминала.

В цикле `while` из листинга 16.2 символ, который возвращается функцией `getchar`, сначала присваивается переменной `c` и затем сравнивается с символом, скрытым за

аббревиатурой EOF. Если равенство выполняется, то это говорит о том, что все символы файла были считаны.

Необходимо обратить внимание на один важный момент, связанный со значением, присвоенным аббревиатуре EOF, которое возвращается функцией `getchar`. На самом деле функция возвращает значение типа `int`, а не `char`. Это происходит потому, что значение для конца файла должно быть уникальным и не должно равняться ни одному из значений, которое в обычном состоянии может возвратить функция `getchar`. Поэтому все значения, возвращаемые функцией `getchar`, присваиваются переменной типа `int`, а не `char`. При этом все работает отлично, поскольку в языке программирования C разрешено присваивать символы переменным типа `int`, хотя это и нельзя считать хорошим стилем программирования.

Но если сохранять значения, возвращаемые функцией `getchar`, в переменной типа `char`, то результат не будет таким предсказуемым. В системах, которые выполняют знаковое расширение символов, такой код может работать довольно хорошо, но в системах, где нет знакового расширения символов, вы можете попасть в бесконечный цикл.

В конце этого раздела хочу еще раз напомнить о том, что сохранять значения, возвращаемые функцией `getchar`, необходимо в переменной типа `int` для того, чтобы в любой ситуации четко фиксировать конец файла.

Тот факт, что вы можете выполнить присваивание в составе условного выражения для цикла `while`, свидетельствует о развитых средствах составления выражений в языке программирования C. Круглые скобки необходимы, поскольку оператор присваивания имеет более низкий приоритет, чем оператор неравенства.

Функции для работы с файлами

Очень может быть, что в большинстве программ, которые вы будете разрабатывать, для выполнения операций ввода-вывода вполне хватит функций `getchar`, `putchar`, `scanf` и `printf`, а также перенаправления ввода-вывода. Однако ситуация изменится, если понадобится выполнить более сложный ввод или вывод в файл. Например, может понадобиться считывать данные из нескольких файлов и производить запись результатов также в несколько различных файлов. Для выполнения такой задачи служат специальные функции, которые рассчитаны только на работу с файлами. Некоторые из этих функций будут описаны в последующих разделах.

Функция `fopen`

Прежде чем выполнять с файлом операции ввода-вывода, его необходимо открыть. Для открытия файла сначала необходимо задать его имя, после чего система проверит наличие такого файла на диске. Если файла на диске нет, то в некоторых ситуациях система создаст файл с таким именем. Когда файл открыт, вы должны указать системе, какого типа операции ввода-вывода вы будете производить с файлом. Если вы открываете файл только для того, чтобы считывать из него данные, то вы должны открыть файл в режиме *только для чтения* (`read mode`).

Если вы хотите производить только запись в файл, то вы должны открыть файл в режиме *только для записи* (`write mode`). Наконец, если вы хотите добавлять информацию в конец файла, в котором уже записаны некоторые данные, то вы открываете файл в режиме *добавления данных* (`append mode`). В последних двух случаях (запись и добавление данных), если указанный файл не существует, то система автоматически создаст для вас новый файл. В режиме *только для чтения* если файл не существует, будет сгенерирована ошибка.

Поскольку в программе может понадобиться открыть несколько файлов одновременно, то должен быть способ идентификации каждого файла на тот случай, если вы будете выполнять операции ввода-вывода только с этим файлом. Для идентификации файлов используются *указатели на файл* (file pointer).

Для того чтобы получить указатель на файл, используется функция `fopen` из стандартной библиотеки, которая открывает необходимый файл и возвращает уникальный указатель, который затем и используется для идентификации файла. Эта функция принимает два аргумента: первый представляет собой строковую константу, задающую имя файла, который должен быть открыт, второй аргумент также является строковой константой, которая задает режим, в котором должен быть открыт файл. Функция возвращает указатель на файл, который будут использовать другие функции для идентификации данного файла.

Если по какой-либо причине файл не может быть открыт, то функция возвращает значение `NULL`, которое определено в заголовочном файле `stdio.h` и в стандартном файле определений `stddef.h`. Для сохранения в программе значения, возвращаемого функции `fopen`, необходимо объявить переменную типа "указатель на файл".

Если учесть все предыдущие замечания, то последовательно записанные утверждения

```
#include <stdio.h>
FILE *inputFile;
inputFile = fopen ("data", "r");
```

приведут к тому, что в системе будет открыт файл с именем `data` в режиме *только для чтения*. Для получения режима *только для записи* необходимо использовать строковую константу `"w"`, а для режима *добавления данных* надо использовать строковую константу `"a"`. Функция `fopen` возвратит указатель на открытый файл, который будет присвоен переменной `inputFile` типа "указатель на файл".

Затем необходимо проверить указатель на тот случай, если файл не был открыт, с помощью следующей последовательности утверждений

```
if ( inputFile == NULL )
    printf ("***Невозможно открыть файл.\n");
else
    // Чтение данных из файла.
```

и убедиться, что данные из файла можно считывать, а всегда указатель на файл не имеет значение `NULL`. Использование указателя со значением `NULL` приведет к непредсказуемым результатам.

Часто вызов функции `fopen`, присваивание результата указателю на файл и проверка на успешное открытие файла выполняются в одном условном выражении.

```
if ( (inputFile = fopen ("data", "r")) == NULL )
    printf ("***Невозможно открыть файл.\n");
```

Функция `fopen` поддерживает еще три режима работы файлов, называемых *режимами обновления* (`"r+"`, `"w+"` и `"a+"`). Все три режима обновления файла разрешают как чтение, так и запись в файл. Режим обновления чтения (`"r+"`) открывает существующий файл как для чтения, так и для записи данных. Режим обновления записи (`"w+"`) подобен режиму *только для записи* (если файл уже существует, то его содержимое уничтожается, если не существует, то файл создается вновь), но при этом разрешены как операции записи, так и чтения. Режим обновления дополнения (`"a+"`), открывает уже существующий файл или создает новый файл, если файл с указанным именем не существует. При этом

считывать данные можно с любого места файла, но записывать данные можно только в конец файла.

В таких операционных системах, как Windows, которые различают текстовые и двоичные файлы, буква `b` может добавляться в конце строки для режима для чтения или записи в двоичный файл. Если вы забудете это сделать, то получите странные результаты, хотя ваша программа будет работать правильно. Это происходит потому, что система по-разному будет обрабатывать последовательность символов возврата каретки и перехода на новую строку при чтении и записи текстов в файл.

Более того, при вводе из файла, встреча с комбинацией символов "Ctrl Z", которая может там находиться, приведет к считыванию символа конца файла, если только этот файл не был открыт как двоичный. Поэтому необходимо открывать двоичные файлы для чтения следующим образом.

```
inputFile = fopen ("data", "rb");
```

Функции `getc` и `putc`

Функция `getc` позволяет считывать один символ из файла. Работа этой функции аналогична работе функции `getchar`, которая уже была описана выше. Единственное различие заключается в том, что функция `getc` принимает один аргумент: указатель на файл, из которого должны считываться данные. Поэтому если предварительно была вызвана функция `fopen`, как показано выше, то выполнение утверждения

```
c = getc (inputFile);
```

приведет к тому, что будет считан один символ из файла с именем `data`. Для считывания последовательности символов необходимо для каждого очередного символа производить вызов функции `getc`. Функция `getc` возвращает значение EOF при достижении конца файла и, как и для функции `getchar`, возвращаемое значение должно сохраняться в переменной типа `int`.

Как вы уже можете предположить, функция `putc` также работает аналогично функции `putchar`, но принимает два аргумента вместо одного. Первым аргументом функции `putc` является символ, который должен быть записан в файл. Вторым аргументом является указатель на файл. Поэтому следующий вызов

```
putc ('\n', outputFile);
```

запишет символ новой строки, на который ссылается указатель на файл `outputFile`. Разумеется, указанный файл должен быть предварительно открыт в режиме *только для записи* или *добавления данных* (или в любом из режимов обновления), с тем чтобы этот вызов был сделан успешно.

Функция `fclose`

Есть операция, которую необходимо выполнять для каждого открытого файла, — это операция закрытия файла. В этом смысле функция `fclose` является противоположностью функции `fopen`. Эта функция сообщает системе, что программа больше не нуждается в определенном файле. При закрытии файла система выполняет необходимые действия по закрытию файла, такие как запись в файл данных, которые могут оставаться в буфере и разъединение связанного с файлом идентификатора. После того как файл будет закрыт, в него нельзя будет ничего записывать и считывать, пока он не будет открыт вновь.

Когда вы закончите все операции с файлом, то не забывайте закрыть файл. Если программа закончится без ошибок, то система сама выполнит закрытие всех связанных с программой файлов. Но лучше всегда самому предусмотреть все действия по закрытию файлов. При этом вы можете избежать недоразумений, связанных с тем, что система предоставляет ограниченное количество файлов для отдельной программы, т.е. одновременно можно открыть только определенное количество файлов. А закрывая ненужные файлы, вы освобождаете место для других файлов и не превышаете выделенный лимит.

Кстати, аргументом для функции `fclose` является указатель на файл, который должен быть закрыт. Поэтому следующий вызов функции

```
fclose (inputFile);
```

приведет к тому, что будет закрыт файл, на который ссылается указатель `inputFile`.

Используя функции `fopen`, `putc`, `getc` и `fclose`, вы можете написать программу, в которой производится копирование одного файла в другой. Программа из листинга 16.3 запрашивает у пользователя имена файлов, из которых производится копирование и в которые производится копирование. Эта программа основывается на программе из листинга 16.2. Вы можете обратиться к этой программе и оценить сделанные изменения.

Предположим, что следующие три строки текста были предварительно записаны в файл с именем `copyme`.

```
This is a test of the file copy
that we have just developed using the
fopen, fclose, getc, and putc functions.
```

Листинг 16.3. Копирование файлов

```
// Программа копирования файлов.
#include <stdio.h>
int main (void)
{
    char inName[64], outName[64];
    FILE *in, *out;
    int c;

    // Получить от пользователя имена файлов.
    printf ("Введите имя файла для копирования: ");
    scanf ("%63s", inName);
    printf ("Введите имя выходного файла: ");
    scanf ("%63s", outName);
    // Открыть входной и выходной файлы.
    if ( (in = fopen (inName, "r")) == NULL ) {
        printf ("Нельзя открыть %s для чтения.\n", inName);
        return 1;
    }
    if ( (out = fopen (outName, "w")) == NULL ) {
        printf ("Нельзя открыть %s для записи.\n", outName);
        return 2;
    }
    // Копирование.
    while ( (c = getc (in)) != EOF )
        putc (c, out);
}
```

```
// Закрывать открытые файлы.
fclose (in);
fclose (out);
printf ("Файл скопирован.\n");

return 0;
}
```

Листинг 16.3. Вывод

Введите имя файла для копирования: соруме
Введите имя выходного файла: here
Файл скопирован.

После копирования проверьте содержимое файла here. В нем должны находиться те же самые три строки текста, которые были в исходном файле соруме.

Функция `scanf`, к которой было произведено обращение в начале программы, позволяет ввести только 63 символа. Этого должно хватить на ввод массивов символов, составляющих имена входного и выходного файлов.

После этого в программе открываются заданные входной и выходной файлы для чтения и записи. Если выходной файл уже существует и открывается только для записи, то в большинстве операционных систем он будет перезаписан.

Если любая из функций `fopen` не завершится успешно, то программа отобразит соответствующее сообщение и выполнение программы будет закончено, причем в систему будет возвращено ненулевое значение свидетельствующее об ошибке. В противном случае, если все будет сделано без ошибок, то начнется копирование одного файла в другой с помощью функций `getc` и `putc`, пока не будет достигнут конец исходного файла.

После окончания копирования оба файла будут закрыты, появится соответствующее сообщение и в систему будет возвращено значение 0, свидетельствующее об успешном завершении процесса.

Функция `feof`

В целях проверки наличия не считанных данных в файле, используется функция `feof`. Единственным аргументом для этой функции является указатель на файл. Функция возвращает целочисленное значение, которое не будет равно 0, если предпринимается попытка производить считывание после того, как достигнут конец файла, и нулевое значение — в противном случае. Поэтому утверждение

```
if ( feof (inFile) )
{
    printf ("Нет данных.\n");
    return 1;
}
```

отобразит сообщение "Нет данных", если все данные из файла, на который ссылается указатель `inFile`, уже были считаны.

Помните, что функция `feof` предупреждает о том, что вы предпринимаете попытку считывать данные, уже после того, как был считан символ конца файла, но не в тот момент, когда вы считываете символ конца файла. Поэтому для того, чтобы функция возвратила ненулевое значение, необходимо считать символ конца файла.

Функции `fprintf` и `fscanf`

Функции `fprintf` и `fscanf` выполняют операции, аналогичные тем, что выполняют функции `printf` и `scanf`, но только для файлов. Эти функции принимают дополнительные аргументы, которые являются указателями на файл, в который должны записываться или из которого должны считываться данные. Поэтому чтобы записать строку "Программирование забавно" в файл, на который ссылается указатель `outFile`, необходимо записать следующее утверждение.

```
fprintf (outFile, "Листингming in C is fun.\n");
```

Аналогично, чтобы считать вещественное число из файла, на который ссылается указатель `inFile`, в переменную `fv` необходимо записать следующее.

```
fscanf (inFile, "%f", &fv);
```

Как и для функции `scanf`, функция `fscanf` возвращает количество аргументов, которые были успешно считаны и значения которых были присвоены переменным, или значение EOF, если достигнут конец файла еще до того, как все операции по присваиванию значений были закончены.

Функции `fgets` и `fputs`

Для считывания целых строк текста из файла и записи в файл, используются функции `fputs` и `fgets`. Функция `fgets` вызывается следующим образом.

```
fgets (buffer, n, filePtr);
```

Здесь переменная `buffer` ссылается на массив символов, в котором должна быть сохранена считанная строка, переменная `n` определяет целое число, представляющее количество символов, которые могут быть сохранены в массиве символов, и указатель ссылается на файл, из которого должна быть считана строка текста.

Функция `fgets` считывает символы из файла до тех пор, пока не встретится символ новой строки, который будет сохранен в массиве, или пока не будет считано `n-1` символов. Функция автоматически размещает символ `null` после последнего символа в буфере. Функция возвращает значение указателя на массив символов, если считывание прошло успешно, и значение `NULL`, если произошла ошибка считывания или была предпринята попытка считывания после символа конца файла.

Функция `fgets` может работать совместно с функцией `sscanf` (см. приложение Б) для более упорядоченного выполнения операций чтения строк текста и для достижения большей степени контроля, чем при использовании только функции `scanf`.

Функция `fputs` записывает строку символов в указанный файл. Функция вызывается следующим образом.

```
fputs (buffer, filePtr);
```

Символы, находящиеся в массиве, на который ссылается указатель `buffer`, будут переписываться в файл, на который ссылается указатель `filePtr`, до тех пор, пока не встретится символ `null` (окончание данных). Этот символ в файл не записывается.

Также есть аналогичные функции с именами `gets` и `puts`, которые используются для чтения и вывода строк текста в окно терминала. Эти функции описаны в приложении Б.

Функции `stdin`, `stdout` и `stderr`

Когда выполняется программа на языке программирования C, то система автоматически открывает три файла для использования в программе. На эти файлы ссылаются три константных указателя типа `FILE`: `stdin`, `stdout` и `stderr`, которые описаны в заголовочном файле `stdio.h`. Указатель `stdin` ссылается на стандартный входной поток, который обычно связан с окном терминала.

Все стандартные функции ввода-вывода, в которых не используется указатель на файл, по умолчанию будут использовать ввод из потока `stdin`. Например, функция `scanf` считывает данные из входного потока `stdin`, и для того, чтобы использовать функцию `fscanf` для аналогичных целей, в ней необходимо указать в качестве первого аргумента именно этот поток. Поэтому при вызове функции

```
fscanf (stdin, "%i", &i);
```

будет считано очередное целое число из стандартного входного потока, которым обычно является окно терминала. Если при этом вход вашей программы был перенаправлен в какой-либо файл, то очередное значение будет считано из этого файла.

Как можно догадаться, указатель `stdout` ссылается на стандартный выходной поток, которым обычно является поток, связанный с окном терминала. Поэтому если написать

```
printf ("hello there.\n");
```

то это будет эквивалентно использованию функции `fprintf` с указанием потока `stdout` в качестве первого аргумента.

```
fprintf (stdout, "hello there.\n");
```

Указатель на файл `stderr` ссылается на стандартный файл ошибок. В этот файл записываются ошибки, происходящие во время выполнения программы. Этот файл также обычно связан с окном терминала. Причина, по которой существует отдельный поток для сообщений об ошибках, заключается в том, что все ошибки должны регистрироваться отдельно от стандартного выходного потока. Причем если выходной поток будет перенаправлен в файл, то ошибки все равно будут отображаться на экране терминала.

Вы можете создавать свои собственные сообщения об ошибках и направлять их в выходной поток `stderr` для того, чтобы они в любом случае отображались на экране. Например, следующее сообщение можно создать с помощью функции `fprintf`:

```
if ( (inFile = fopen ("data", "r")) == NULL )
{
    fprintf (stderr, "Нельзя открыть файл для чтения.\n");
    ...
}
```

При этом указанное сообщение об ошибке будет записано в стандартный поток ошибок `stderr`, если файл с именем `data` не будет нормально открыт для чтения. Даже если стандартный выходной поток будет перенаправлен в файл, это сообщение все же отобразится на экране.

Функция exit

Иногда приходится принудительно заканчивать выполнение программы, например, когда в результате проверки некоторого условия была обнаружена ошибка. Вы уже знаете, что выполнение программы заканчивается автоматически, когда выполняется последнее утверждение из функции `main` или когда в этой функции встречается утверждение `return`. Для того чтобы явно прервать выполнение программы, помимо этих случаев, необходимо использовать функцию `exit`. Вызов функции выполняется следующим образом.

```
exit (n);
```

При этом прекращается выполнение текущей программы (той программы, где встретилось это утверждение). Все открытые файлы будут автоматически закрыты системой. Целое число `n`, которое используется в качестве аргумента, называется *состоянием выхода* (`exit status`) и выполняет ту же самую функцию, что и значение, возвращаемое функцией `main`.

В стандартном заголовочном файле `stdlib.h` описана переменная `EXIT_FAILURE`, которая представляет собой целое число и которую можно использовать для того, чтобы указать на завершение программы с ошибкой, и переменная `EXIT_SUCCESS`, которая свидетельствует об успешном выполнении программы.

Когда программа завершается при достижении последнего утверждения в функции `main`, ее состояние выхода считается неопределенным. Если в программе необходимо использовать значение состояния выхода, то вы не должны допускать, чтобы выход из программы происходил подобным образом. Необходимо сделать так, чтобы программа заканчивалась либо утверждением `exit`, либо `return` в функции `main`.

В качестве примера использования функции `exit` рассмотрим функцию, которая приводит к завершению программы с возвращаемым значением `EXIT_FAILURE`, если файл, указанный в качестве аргумента, не может быть открыт для чтения. Хотя обычно в таких случаях возвращается сообщение о неудачном открытии файла, а не прекращается выполнение программы.

```
#include <stdlib.h>
#include <stdio.h>
FILE *openFile (const char *file)
{
    FILE *inFile;

    if ( (inFile = fopen (file, "r")) == NULL )
    {
        fprintf (stderr, "Нельзя открыть %s для чтения.\n", file);
        exit (EXIT_FAILURE);
    }

    return inFile;
}
```

Помните, что нет существенной разницы между завершением программы с помощью функций `exit` или `return`, встретившихся в функции `main`. Они обе прерывают выполнение и возвращают состояние выхода в вызвавшую программу. Основное отличие проявляется тогда, когда эти функции встречаются не в функции `main`. При этом функция `exit`, в любом случае, приводит к немедленному завершению программы, тогда как функция `return` просто возвращает значение в вызвавшую подпрограмму.

Переименование и перемещение файлов

Функцию `rename` из стандартной библиотеки можно использовать для изменения имен файлов. Она принимает два аргумента: старое и новое имя файла. Если по некоторым причинам переименование не состоится (например, если неверно указано имя файла или система не может его переименовать), то функция `rename` возвращает ненулевое значение. Следующие утверждения

```
if ( rename ("tempfile", "database") )
{
    fprintf (stderr, "Can't rename tempfile\n");
    exit (EXIT_FAILURE);
}
```

приведут к переименованию файла с именем `tempfile` в файл с именем `database` и произведут проверку правильности выполнения этой операции.

Функция `remove` стирает файл, который указан в качестве аргумента. Она возвращает ненулевое значение, если стирание файла произошло успешно. Следующие строки

```
if ( remove ("tempfile") )
{
    fprintf (stderr, "Can't remove tempfile\n");
    exit (EXIT_FAILURE);
}
```

приведут к попытке стереть файл с именем `tempfile` и произведут запись сообщений об ошибке в стандартный поток ошибок или к попытке выхода из программы — если файл стереть не удалось.

Кстати, если вы хотите узнать больше об использовании функции `pererror` из стандартной библиотеки ввода-вывода для получения отчета об ошибках, обратитесь к приложению Б.

На этом заканчивается обсуждение операций ввода-вывода в языке С. Как уже упоминалось, из-за ограничений по объему очень многие функции из стандартных библиотек не рассмотрены в этой книге. Стандартная библиотека языка С содержит большое количество функций для выполнения операций со строками символов, для получения последовательности случайных чисел, для математических вычислений и для динамического управления памятью. В приложении Б рассказывается о многих из этих функций.

Упражнения

1. Наберите и выполните три программы, представленные в этой главе. Сравните вывод, полученный вами в результате выполнения программ, с тем, что представлен в книге.
2. Вернитесь к более ранним разработкам программ из этой книги и поэкспериментируйте с перенаправлением ввода-вывода в файлы.
3. Напишите программу для копирования одного файла в другой, заменяя при этом все строчные буквы на заглавные.
4. Напишите программу, которая объединяет соответствующие строки из двух файлов и записывает результат в поток `stdout`. Если в одном файле строк меньше, чем в другом, то оставшиеся строки просто копируются в выходной файл.
5. Напишите программу, которая записывает колонки от `m` до `n` из каждой строки в стандартный поток `stdout`. Программа должна запрашивать значения `m` и `n`.
6. Напишите программу, которая отображает в окне терминала по 20 строк из файла. После отображения первых 20 строк программа должна ждать ввода символа с терминала. Если введенным символом будет буква `q`, то программа должна прекратить отображение файла. Все другие введенные символы должны вызывать последующий вывод очередных 20 строк.

Дополнительные ВОЗМОЖНОСТИ

В этой главе рассказывается о таких возможностях языка программирования C, как аргументы командной строки и распределение памяти.

Еще два утверждения

В этом разделе будут обсуждаться два утверждения языка программирования C, о которых до этого момента вы еще ничего не знали, — это утверждения `goto` и `null`.

Утверждение `goto`

Все, кто изучал структурное программирование, знают о плохой репутации утверждения `goto`. Но все компьютерные языки явно или неявно используют это утверждение.

Выполнение утверждения `goto` приводит к переходу на обработку фрагмента программы, который нарушает последовательность выполняемых команд и может находиться отдельно от основного блока программы. Переход на выполнение этого фрагмента совершается безусловно после утверждения `goto`. Для идентификации необходимого фрагмента используется метка. Метка — это имя, сформированное по тем же самым правилам, которые применяются к идентификаторам. Непосредственно за меткой должно располагаться двоеточие. Метка размещается непосредственно перед тем фрагментом программы, на который будет сделан переход. Например, утверждение

```
goto out_of_data;
```

приведет к непосредственному переходу на фрагмент программы, который начинается с метки.

```
out_of_data:
```

Эта метка может располагаться в любом месте программы, до или после утверждения `goto`, и должна использоваться, как показано в следующем утверждении.

```
out_of_data: printf ("Unexpected end of data.\n");  
...
```

Программисты из-за своей лени часто неправильно используют утверждения `goto` для перехода на другой фрагмент программы. Утверждение `goto` прерывает последовательный поток утверждений, в результате чего программа становится сложнее. Использование большого количества утверждений `goto` приводит к тому, что программу становится невозможно понять. По этой причине использование утверждений `goto` считается плохим стилем программирования.

Утверждение `null`

В языке программирования C можно ставить точку с запятой там, где должно располагаться утверждение. Такое использование точки с запятой имеет эффект утверждения `null`, или пустого утверждения, которое ничего ни делает. Хотя это и кажется абсолютно бесполезным, но такое утверждение часто используется с конструкциями `while`, `for` и `do`. Например, целью следующего утверждения является сохранение всех считанных при стандартном вводе символов в символьном массиве с именем `text`. Символом окончания ввода считается символ новой строки.

```
while ( (*text++ = getchar ()) != '\n' )
;
```

Здесь все операции выполняются в выражении условия для цикла `while`. Поэтому здесь необходимо утверждение `null`, поскольку компилятор воспринимает утверждение, следующее за заголовком цикла, как тело цикла. Без утверждения `null` выполняться будет следующее за циклом утверждение, что абсолютно неприемлемо. Поэтому возникает необходимость в утверждении `null`, которое и будет заменять тело цикла, после чего компилятор правильно оттранслирует программу.

В следующем цикле `for` производится копирование символов из стандартного входа в стандартный выход до тех пор, пока не встретится конец файла.

```
for ( ; (c = getchar ()) != EOF; putchar (c) )
;
```

В следующем цикле `for` подсчитывается количество символов, которые находятся в стандартном входе.

```
for ( count = 0; getchar () != EOF; ++count )
;
```

Рассмотрим еще один вариант использования утверждения `null`. В следующем утверждении символьная строка `from` копируется в строку `to`.

```
while ( (*to++ = *from++) != '\0' )
;
```

У читателей может сложиться впечатление, что я советую вкладывать как можно больше функциональности в условные выражения для циклов `while` или `for`. Наоборот, старайтесь так не делать. Только простые выражения необходимо использовать в качестве условных выражений для циклов. Всегда лучше формировать тело цикла, если только это не связано с "выжиманием" высокой производительности из программы.

Предыдущее утверждение `while` лучше записать следующим образом:

```
while ( *from != '\0' )
*to++ = *from++;
*to = '\0';
```

Это сделает программу более читабельной, и будет проще разобраться в логике работы цикла.

Объединения

Одной из довольно редко используемых конструкций языка программирования C являются *объединения*. Эти конструкции в основном используются при необходимости сохранения различных типов данных в одном и том же пространстве памяти. Например, если вы хотите задать одну переменную с именем `x`, которая будет использоваться для хранения одного символа, вещественного числа или целочисленного значения, вы должны определить объединение, например, с именем `mixed`.

```
union mixed
{
    char c;
    float f;
    int i;
};
```

Синтаксис объявления объединения аналогичен объявлению структуры, за исключением ключевого слова `union`, которое ставится вместо слова `struct`. Но реальное отличие структуры от объединения проявляется в том, как они распределяют память. Если объявить переменную типа `union mixed`, как это сделано ниже

```
union mixed x;
```

то это не значит, что в памяти будет выделено место для трех членов `c`, `f` и `i`, к которым можно обратиться. Будет выделено место только для одной переменной, к которой можно обратиться по имени `c`, `f` и `i`. Таким образом, переменная `x` может использоваться только для хранения типа `char`, `float` или `int`, но не для хранения всех типов вместе. Можно сохранить символ в переменной `x` с помощью следующего утверждения.

```
x.c = 'K';
```

Сохраненный символ можно впоследствии извлечь аналогичным образом. Следовательно, для того чтобы отобразить этот символ в окне терминала, можно использовать следующее утверждение.

```
printf ("Character = %c\n", x.c);
```

Для сохранения в переменной `x` вещественного значения, можно написать следующее.

```
x.f = 786.3869;
```

Наконец, для помещения в переменную `x` результата деления целочисленного значения переменной `count` на 2, надо будет написать следующее утверждение.

```
x.i = count / 2;
```

Поскольку типы `float`, `char` и `int`, хранящиеся в переменной `x`, находятся в одном и том же пространстве памяти, они не могут храниться одновременно. Более того, вы не знаете, какой тип хранится в переменной `x`, т.е. нет никакой информации о том, значение какого типа переменной было сохранено в переменной `x` типа *объединение*.

Тип члена объединения переменной определяется согласно тому, как к ней будет производиться обращение. Например, выражение

```
x.i / 2
```


будет рассчитываться по правилам целочисленной арифметики, поскольку оба операнда, как `x.i`, так и `2`, имеют тип `int`.

В объединении может располагаться сколь угодно много членов. Компилятор языка программирования C выделит достаточно памяти для хранения самого большого члена объединения. Структуры также могут в качестве членов содержать структуры, и в случае с массивами. При объявлении массива можно не указывать его имя. Соответствующая переменная может быть объявлена одновременно с объявлением объединения. Также можно объявлять указатели на объединение, правила работы с которыми аналогичны правилам работы со структурами.

Вполне можно инициализировать любой член структуры. Если имя члена структуры не указано, то используется первый член из списка членов структуры, поэтому в выражении

```
union mixed x = { '#' };
```

будет инициализирован первый член объединения `x`, который имеет тип `char` и которому будет присвоено значение `'#'`.

Но в случае указания имени, любой член объединения можно инициализировать следующим образом.

```
union mixed x = { .f = 123.456; };
```

При этом член объединения `f`, хранящий вещественные значения, будет инициализирован значением `123.456`.

Переменная типа объединение может быть инициализирована переменной аналогичного типа.

```
void foo (union mixed x)
{
    union mixed y = x;
    ...
}
```

Здесь в функции `foo` производится присваивание объединению `y` значения объединения `x`, передаваемого в качестве аргумента. Использование объединения позволит создать массив, в элементах которого можно хранить значения различных типов. Например, утверждения

```
struct
{
    char *name;
    enum symbolType type;
    union
    {
        int i;
        float f;
        char c;
    } data;
} table [kTableEntries];
```

задают массив с именем `table`, содержащий `kTableEntries` элементов.

Каждый элемент массива является структурой, состоящей из указателя на символы с именем `name`, перечисления с именем `type` и объединения с именем `data`. Каждый член массива типа `data` может содержать или тип `int`, или тип `float`, или `char`. Член

массива типа `type` может использоваться для хранения типа, содержащегося в объединении `data`. Например, можно использовать слово `"INTEGER"` — если хранится тип `int`, слово `"FLOATING"`, если хранится тип `float`, и слово `"CHARACTER"`, если хранится тип `char`. Этой информации будет достаточно для того, чтобы знать, как обращаться к отдельному члену типа `data` для отдельного элемента массива.

Для сохранения символа `"#"` в элементе массива `table[5]`, и последующей установки поля `type` для указания на то, что символ сохранен, необходимо использовать следующие утверждения.

```
table[5].data.c = '#';
table[5].type = CHARACTER;
```

Теперь при просмотре элементов массива `table` можно легко определить тип данных, сохраненных в каждом элементе, производя анализ поля `type`. Например, в следующем цикле будут отображены название типа и связанное с ним значение элементов массива `table` в окне терминала.

```
enum symbolType { INTEGER, FLOATING, CHARACTER };
...
for ( j = 0; j < kTableEntries; ++j )
{
    printf ("%s ", table[j].name);
    switch ( table[j].type )
    {
        case INTEGER:
            printf ("%i\n", table[j].data.i);
            break;
        case FLOATING:
            printf ("%f\n", table[j].data.f);
            break;
        case CHARACTER:
            printf ("%c\n", table[j].data.c);
            break;
        default:
            printf ("Unknown type (%i), element %i\n", table[j].type, j);
            break;
    }
}
```

Аналогичную конструкцию можно использовать для хранения таблицы идентификаторов, в которую можно записать как имя каждого идентификатора, его тип и значение, так и другую необходимую информацию.

Оператор “запятая”

Не зная синтаксиса языка, трудно предположить, что запятая может использоваться в выражениях в качестве оператора. Оператор “запятая” используется в т.н. “каскадных вычислениях”. В главе 5, “Программирование циклов”, вы узнали, что при использовании цикла `for` можно включать более одного выражения в любое из полей, выделяя каждое выражение с помощью запятой. Например, заголовок цикла `for`

```
for ( i = 0, j = 100; i != 10; ++i, j -= 10 )
```

```
***
```

инициализирует переменную *i* значением 0 и переменную *j* значением 100 еще до того, как цикл начнет выполняться. В процессе итераций будет инкрементироваться значение переменной *i* и уменьшаться на 10 значение переменной *j*.

Оператор “запятая” может использоваться для разделения нескольких выражений в любом месте, где по правилам языка программирования C может использоваться одно выражение. Разделенные запятой выражения выполняются слева направо. Поэтому в цикле

```
while ( i < 100 )  
sum += data[i], ++i;
```

будет производиться сложение элемента массива *data[i]* с переменной *sum* и затем будет инкрементировано значение переменной *i*. Обратите внимание, что в данном случае нет необходимости ставить фигурные скобки, поскольку считается, что за заголовком цикла следует только одно утверждение, состоящее из двух выражений, разделенных оператором “запятая”.

Поскольку в результате всех операций в языке программирования C должно возвращаться значение, то оператор “запятая” возвращает результат вычисления правого выражения.

Но при этом обратите внимание на то, что запятая используется и для разделения аргументов при вызове функции или разделении имен переменных в списке объявлений. Это разные синтаксические конструкции, и их нельзя рассматривать как пример использования запятой в качестве оператора.

Квалификаторы

Следующие квалификаторы можно использовать непосредственно перед переменными для передачи компилятору информации о предпочтительном использовании переменной для получения более эффективного кода.

Квалификатор `register`

Если в некоторой функции отдельная переменная используется особенно активно, то необходимо обеспечить как можно более быстрый доступ к этой переменной. Обычно для этого переменную необходимо хранить в одном из машинных регистров при выполнении этой функции. Это требование можно передать компилятору при объявлении переменной, поставив перед ней ключевое слово `register`, как показано ниже.

```
register int index;  
register char *textPtr;
```

Эти локальные переменные и формальные параметры могут объявляться с помощью ключевого слова `register`. Типы переменных, которые могут сохраняться в регистрах, отличаются на разных машинах. Базовые типы данных обычно хранятся в регистрах, как и указатели на любые типы данных. Но даже если вы объявили переменную с ключевым словом `register`, это не означает, что все будет сделано в соответствии с вашим требованием. Все зависит от компилятора, т.е. в некоторых случаях он будет обращаться с регистровыми переменными как с обычными переменными.

Необходимо также отметить, что регистровые переменные нельзя использовать с оператором адресации.

Квалификатор `volatile`

Этот тип квалификатора является противоположностью к квалификатору `const`. При его использовании компилятору явно указывается, что значение данной переменной будет изменяться. Это необходимо в тех случаях, когда компилятор попытается оптимизировать код и удалить переменную, создав более компактно выражение в предположении, что этой переменной не будет присваиваться значение. В качестве примера рассмотрим порты ввода-вывода. Предположим, что вы объявили выходной порт в программе, на который указывает переменная с именем `outPort`. Если вы хотите записать в порт два символа, например символы `O` и `N`, то вы должны написать следующие утверждения.

```
*outPort = 'O';
*outPort = 'N';
```

При этом достаточно “разумный” компилятор заметит, что два последовательных утверждения помещают значение в одну и ту же переменную. Поскольку никаких изменений после первого присваивания с этой переменной не происходит, то компилятор зафиксирует ошибку. Во избежание этого необходимо объявить переменную `outPort` с квалификатором `volatile`, как показано ниже.

```
volatile char *outPort;
```

Квалификатор `restrict`

Подобно квалификатору `register`, квалификатор `restrict` указывает компилятору на необходимость оптимизации. Как и ранее, компилятор может принять или отвергнуть это предложение. Это квалификатор используется указания компилятору на то, что отдельный указатель является единственной ссылкой (косвенной или непосредственной) на значение в его области видимости. Это означает, что на это значение не может ссылаться никакой другой указатель или переменная в этой области видимости.

Строки

```
int * restrict intPtrA;
int * restrict intPtrB;
```

сообщают компилятору о том, что в области видимости, где объявлены переменные `intPtrA` и `intPtrB`, они не будут изменены. Они используются для ссылки на целочисленные значения, которые являются, например, взаимоисключающими.

Аргументы командной строки

Во многих случаях разработанная программа требует, чтобы пользователь вводил небольшую информацию с экрана терминала. Эта информация может представлять собой число, которое, например, необходимо использовать в расчетах, или слово, которое вы хотите найти в словаре.

Вместо того, чтобы создавать в программе запрос на ввод данных, можно просто вводить информацию в программу в момент запуска программы на выполнение. Такая возможность широко используется и связана с *аргументами командной строки*.

Как уже не раз упоминалось, единственной отличительной особенностью функции `main` является то, что это имя известно среде выполнения. Именно с него начинается выполнение программы. Система запуска обращается к функции `main` точно так же,

как вызывается любая функция в программе на языке C. После выполнения функции `main`, контроль возвращается к среде выполнения, которая может проконтролировать выполнение программы.

Когда функция `main` вызывается средой исполнения, в нее передаются два аргумента. Первый аргумент, который обычно обозначают как `argc` (от слов *argument count* — счетчик аргументов), является целым числом, которое определяет количество аргументов, введенных с командной строки.

Вторым аргументом для функции `main` является массив символьных указателей, который называется `argv` (*argument vector* — вектор аргументов). В этом массиве должно находиться `argc+1` указателей. При этом минимальным значением для `argc` является значение 0. Первый указатель в этом массиве указывает на имя выполняемой программы или на нулевую строку, если имя программы недоступно для среды выполнения. Последующие значения элементов массива указывают на значения, которые находятся в командной строке и используются для инициализации программы. Последний указатель массива `argv`: `argv[argc]`, должен быть равен `null`.

Для доступа к аргументам командной строки, функция `main` должна быть предварительно объявлена как функция с двумя аргументами. По негласному соглашению, это записывается следующим образом

```
int main (int argc, char *argv[])
{
    ...
}
```

Помните, что объявление переменной `argv` задаст массив, в котором содержатся элементы типа “указатель на тип `char`”. Для того чтобы попрактиковаться с аргументами командной строки, обратитесь к программе из листинга 10.10, которая ищет слово в словаре и печатает его значение. Вы можете использовать аргументы командной строки для того, чтобы вводить слово, значение которого необходимо получить, одновременно с запуском программы на выполнение, как показано в следующей команде:

```
lookup aerie
```

Это исключает дополнительное обращение к пользователю для ввода нужного слова, поскольку оно уже введено в командной строке.

Если предыдущая команда будет введена, то система автоматически передаст в функцию `main` указатель на символьную строку “aerie” в элементе массива `argv[1]`. Напомню, что указатель, содержащийся в элементе `argv[0]`, содержит ссылку на имя программы, которым в нашем случае будет “lookup”.

Процедура `main` должна выглядеть следующим образом.

```
#include <stdlib.h>
#include <stdio.h>
int main (int argc, char *argv[])
{
    const struct entry dictionary[100] =
    { { "aardvark", "a burrowing African mammal" },
      { "abyss", "a bottomless pit" },
      { "acumen", "mentally sharp; keen" },
      { "addle", "to become confused" },
      { "aerie", "a high nest" },
      { "affix", "to append; attach" },
```

```

    { "agar", "a jelly made from seaweed" },
    { "ahoy", "a nautical call of greeting" },
    { "aigrette", "an ornamental cluster of feathers" },
    { "ajar", "partially opened" }
};
int entries = 10;
int entryNumber;
int lookup (const struct entry dictionary [], const char search[],
            const int entries);
if ( argc != 2 )
{
    fprintf (stderr, "No word typed on the command line.\n");
    return EXIT_FAILURE;
}
entryNumber = lookup (dictionary, argv[1], entries);
if ( entryNumber != -1 )
    printf ("%s\n", dictionary[entryNumber].definition);
else
    printf ("Sorry, %s is not in my dictionary.\n", argv[1]);
return EXIT_SUCCESS;
}

```

В процедуре `main` выполняется проверка на наличие дополнительных слов после ввода имени программы при запуске программы на выполнение. Если этих слов нет или их количество превышает единицу, то значение аргумента `argc` не будет равно 2. В этом случае программа выводит сообщение об ошибке в стандартный поток ошибок и на терминал, возвращая в среду выполнения состояние `EXIT_FAILURE`.

Если значение аргумента `argc` равно 2, то вызывается функция `lookup` для поиска в словаре слова, на которое указывает элемент множества `argv[1]`. Если слово найдено, то отображается его описание.

В следующем примере с аргументами командной строки, в программе из листинга 16.3 выполняется копирование файлов. Программа из листинга 17.1, который приведен ниже, в качестве аргументов командной строки принимает два имени файлов, и поэтому не производится дополнительное обращение к пользователю для получения имен файлов.

Листинг 17.1. Копирование файлов с использованием аргументов командной строки

```

// Программа копирования файлов – версия 2.
#include <stdio.h>
int main (int argc, char *argv[])
{
    FILE *in, *out;
    int c;

    if ( argc != 3 )
    {
        fprintf (stderr, "Need two files names\n");
        return 1;
    }
    if ( (in = fopen (argv[1], "r")) == NULL )
    {
        fprintf (stderr, "Can't read %s.\n", argv[1]);
    }
}

```

```

    return 2;
}
if ( (out = fopen (argv[2], "w")) == NULL ) {
    fprintf (stderr, "Can't write %s.\n", argv[2]);
    return 3;
}
while ( (c = getc (in)) != EOF )
    putc (c, out);
printf ("File has been copied.\n");
fclose (in);
fclose (out);

return 0;
}

```

В программе сначала производится проверка на наличие двух имен файлов после ввода имени программы. Если эти имена присутствуют, то на имя входного файла будет указывать элемент множества `argv[1]`, а на имя второго – элемент `argv[2]`. После открытия первого файла для чтения и второго файла для записи и после проверки того, что эти операции выполнены успешно, программа символ за символом копирует первый файл во второй, как это делалось и ранее.

Обратите внимание, что в этой программе предусмотрены четыре ситуации для прекращения выполнения программы: неправильное количество аргументов командной строки, невозможность открыть файл для чтения, невозможность открыть файл для записи и успешное выполнение задачи. Помните, что если вы используете в программе возвращаемое значение, то необходимо всегда завершить программу установленной возвращаемой значения. Если этого не сделать, то будет возвращено неопределенное состояние.

Если программа из листинга 17.1 будет называться `copyf` и для запуска программы в командной строке будет набрано следующее

```
copyf foo fool
```

то после входа в функцию `main` массив `argv` будет выглядеть так, как показано на рис. 17.1.

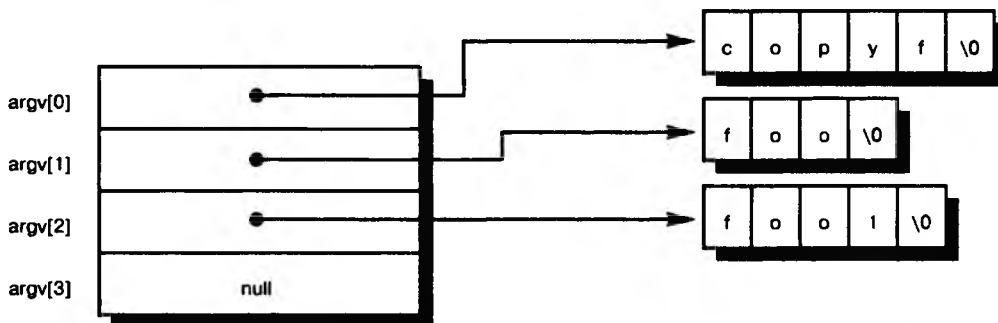


Рис. 17.1. Начальное состояние массива `argv` после запуска программы `copyf`

Помните, что аргументы командной строки всегда сохраняются как символьные строки. Если выполнить программу `power` (степень) с аргументами 2 и 16, для чего следует ввести команду

```
power 2 16
```

то в программе будет сохранен указатель на символьную строку "2" для элемента множества `argv[1]`, и указатель на символьную строку "16" — для `argv[2]`.

Если эти аргументы должны использоваться как целые числа для возведения в степень (что и ожидается от программы), то они должны быть преобразованы в целые числа в самой программе. Несколько программ для проведения таких преобразований находятся в стандартной библиотеке: `sscanf`, `atof`, `atoi`, `strtod` и `strtol`. Они описаны в Приложении Б, "Стандартные библиотеки языка C".

Динамическое распределение памяти

Когда в языке C объявляется переменная, будь это простой тип данных, массив или структура, то при этом всегда резервируется отдельные участки компьютерной памяти для хранения значений, присваиваемых этим переменным. Компилятор автоматически распределяет необходимое количество памяти для хранения значений.

Часто бывает не только желательно, но и необходимо распределять память во время выполнения программы. Предположим, необходимо не только считать данные из файла, но и расположить их в памяти. Но при этом вы не можете заранее, сколько байтов находится в файле, и узнаете об этом только тогда, когда программа будет запущена на выполнение. В этом случае можно выбрать один из трех вариантов.

- Во время компиляции задать массив, который будет содержать максимально возможное количество элементов.
- Использовать массив с переменным размером, который будет подгоняться под размер файла во время выполнения.
- Распределять память динамически с помощью процедур, предусмотренных в языке программирования C.

Для первого случая необходимо задать массив, содержащий максимально возможное количество элементов, которые могут быть считаны из файла. Это можно сделать следующим образом.

```
#define kMaxElements 1000
struct dataEntry dataArray [kMaxElements];
```

После этого вы можете использовать свою программу для работы с файлами, длина которых не превышает 1000 байтов. Но если придется работать с файлами большей длины, то необходимо будет изменить значение переменной `kMaxElements`, для чего надо будет откорректировать исходную программу и вновь откомпилировать ее. Причем это не гарантирует, что в будущем вам вновь не придется изменять.

Во втором случае, если вы знаете количество элементов массива до того, как запустите программу на выполнение (известна длина рабочего файла), то можно задать длину массива следующим образом.

```
struct dataEntry dataArray [dataItems];
```

В этом случае предполагается, что переменная `dataItems` содержит точное количество необходимых элементов массива, способных вместить файл.

Но, используя функции динамического распределения памяти, можно получить именно столько памяти, сколько необходимо. При этом можно распределять память во время выполнения программы. Для того чтобы воспользоваться динамическим распределением памяти, необходимо сначала познакомиться с тремя функциями и одним оператором `new`.

Функции `calloc` и `malloc`

В стандартной библиотеке языка C есть две функции, называемые `calloc` и `malloc`, которые можно использовать для распределения памяти во время выполнения программы. Функция `calloc` принимает два аргумента, которые задают количество элементов для резервирования памяти и размер каждого элемента в байтах. Функция возвращает указатель на начало того места в памяти, где размещены новые элементы. Счетчик количество записанных значений устанавливается в 0.

Функция `calloc` возвращает указатель на тип `void`, который в языке C определяет настраиваемый тип. Перед сохранением значений этот указатель используется для определения указателя соответствующего типа, для чего необходимо использовать оператор приведения типов.

Функция `malloc` работает аналогичным образом, за тем исключением, что она принимает только один аргумент — общее количество байтов, которое необходимо сохранить в памяти, и не устанавливает количество сохраненных байтов в 0.

Функции динамического распределения памяти объявлены в стандартном заголовочном файле `<stdlib.h>`, который необходимо подключить к программе, если вы будете использовать эти функции.

Оператор `sizeof`

Для того чтобы определить количество памяти, зарезервированной с помощью функций `calloc` или `malloc`, в языке программирования C используется оператор `sizeof`. Оператор возвращает размер заданного элемента в байтах. Аргументом для оператора `sizeof` может быть переменная, имя массива, имя базового типа данных, имя унаследованного типа данных или выражение. Например, выражение

```
sizeof (int)
```

возвращает количество байтов, необходимое для хранения значения типа `int`. На компьютерах с процессором Pentium 4 это будет число 4, поскольку для хранения целочисленных переменных в таких компьютерах используется 32 бита.

Если `x` является именем массива для хранения 100 целочисленных значений, то выражение

```
sizeof (x)
```

возвратит размер памяти, требуемый для хранения 100 целых чисел (или число 400, если это компьютер с процессором Pentium 4). Выражение

```
sizeof (struct dataEntry)
```

возвратит количество памяти, требуемой для хранения одной структуры `dataEntry`. Наконец, если переменная `data` объявлена как массив элементов типа `struct dataEntry`, то выражение:

```
sizeof (data) / sizeof (struct dataEntry)
```

возвратит количество элементов массива `data` (переменная `data` должна быть объявлена как массив, а не параметр или ссылка на внешний массив). Выражение

```
sizeof (data) / sizeof (data[0])
```

возвратит то же значение. Макроопределение

```
#define ELEMENTS(x) (sizeof(x) / sizeof(x[0]))
```

упростит использование таких выражений. Применяя это макроопределение, достаточно будет написать что-то вроде

```
if ( i >= ELEMENTS (data) )
...
или
for ( i = 0; i < ELEMENTS (data); ++i )
...
```

Вы должны помнить, что оператор `sizeof` — это именно оператор, а не функция, хотя он и похож на функцию. Этот оператор обрабатывается во время компиляции, а не во время выполнения программы, за исключением случая с использованием массивов переменной длины в качестве аргумента. Если такой массив не используется, то компилятор подсчитывает значение выражения для `sizeof` и возвращает результат расчета, который будет считаться константой.

Используйте оператор `sizeof` во всех возможных случаях, что позволит избежать излишних расчетов в вашей программе.

Вернемся к вопросу о распределении памяти. Если необходимо выделить достаточно памяти в программе для хранения 1000 целых чисел, то вы должны использовать функцию `calloc` следующим образом.

```
#include <stdlib.h>
...
int *intPtr;
...
intPtr = (int *) calloc (sizeof (int), 1000);
```

Использование функции `malloc` для этих же целей можно представить следующим образом.

```
intPtr = (int *) malloc (1000 * sizeof (int));
```

Помните, что обе функции, `malloc` и `calloc`, возвращают указатель на тип `void`, и как уже упоминалось, этот указатель должен быть приведен к указателю на соответствующий тип. В предыдущем примере указатель приводится к указателю на целочисленное значение и затем присваивается переменной `intPtr`.

Если вы запросите памяти больше, чем система может выделить, то функции `calloc` и `malloc` возвратят нулевой указатель. Поэтому при использовании этих функций всегда проводите проверку на наличие нулевого указателя во избежание неприятностей.

В следующем фрагменте кода распределяется память для 1000 указателей на целые числа и производится проверка возвращаемого значения. Если распределение памяти не было выполнено успешно, то программа выводит сообщение об ошибке в стандартный поток ошибок и прекращает выполнение.

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

...
int *intPtr;

...
intPtr = (int *) calloc (sizeof (int), 1000);
if ( intPtr == NULL )
{
    fprintf (stderr, "calloc failed\n");
    exit (EXIT_FAILURE);
}
```

Если произошло успешное выделение памяти, то переменная `intPtr` будет указывать на массив из 1000 целых чисел и ее можно использовать в расчетах. Поэтому для того чтобы установить для всех элементов массива значение -1, можно написать следующее.

```
for ( p = intPtr; p < intPtr + 1000; ++p )
    *p = -1;
```

При этом предполагается, что указатель `p` объявлен как указатель на целое число.

Для того чтобы зарезервировать память для `n` элементов типа `struct dataEntry`, сначала необходимо объявить указатель на соответствующий тип

```
struct dataEntry *dataPtr;
```

и затем вызвать функцию `calloc` для выделения необходимого количества элементов.

```
dataPtr = (struct dataEntry *) calloc (n, sizeof (struct dataEntry));
```

Выполнение предыдущего утверждения происходит следующим образом.

1. Функция `calloc` вызывается с двумя аргументами, первый из которых говорит о том, что необходимо динамически распределить память для `n` элементов, а второй определяет размер каждого элемента.
2. Функция `calloc` возвращает указатель на то место в памяти, где выделена память. Если память не может быть выделена, то возвращается нулевой указатель.
3. Указатель приводится к типу "указатель на `struct dataEntry`" и присваивается указателю `dataPtr`.

И еще раз напомним, что значение указателя `dataPtr` должно быть проверено для подтверждения того, что произошло выделение памяти. Если оно выполнено успешно, то значение будет не нулевым. Затем этот указатель можно использовать обычным образом, т.е. он будет указывать на массив из `n` элементов типа `dataEntry`. Например, если структура `dataEntry` содержит член типа `int` с именем `dataPtr`, то вы можете присвоить ему значение 100 с помощью указателя `dataPtr` следующим образом.

```
dataPtr->index = 100;
```

Функция free

Когда вы заканчиваете работу с памятью, которая была распределена динамически с помощью функций `calloc` или `malloc`, то вы должны вернуть эту память обратно в систему с помощью функции `free`.

Единственным аргументом для этой функции будет указатель на начало выделенной памяти, который возвращается функциями `calloc` или `malloc`. Поэтому утверждение

```
free (dataPtr);
```

возвратит память, которая была ранее распределена с помощью функции `calloc`. При этом предполагается, что указатель `dataPtr` все еще указывает на начало распределенной памяти. А функция `free` ничего не возвращает.

Память, которая была освобождена с помощью функции `free`, в дальнейшем может быть опять использована при вызове функций `calloc` или `malloc`. Для программ, в которых производится значительное выделение памяти, необходимо всегда освобождать неиспользуемую память и не забывать об этом. Также проверьте, что для функции `free` вы указали именно тот указатель, который указывает на начало ранее выделенного участка памяти.

Динамическое распределение памяти всегда используется при работе со связанными структурами, например, со связанными списками. Когда необходимо добавить новый элемент в список, вы должны динамически выделить память для этого элемента с помощью функций `calloc` или `malloc` и связать его со списком с помощью указателя, возвращенного этими функциями. Например, предположим, что указатель `listEnd` указывает на конец однонаправленного списка типа `struct entry`, объявленного следующим образом.

```
struct entry
{
    int value;
    struct entry *next;
};
```

Также определяется функция `addEntry`, которая принимает в качестве аргумента указатель на начало связанного списка и добавляет новый элемент в конец списка.

```
#include <stdlib.h>
#include <stddef.h>
// Добавить новый элемент в конец списка.
struct entry *addEntry (struct entry *listPtr)
{
    // Найти конец списка.
    while ( listPtr->next != NULL )
        listPtr = listPtr->next;
    // Получить память для нового элемента.
    listPtr->next = (struct entry *) malloc (sizeof (struct entry));
    // Добавить null в новый конец списка.
    if ( listPtr->next != NULL )
        (listPtr->next)->next = (struct entry *) NULL;
    return listPtr->next;
}
```

Если распределение памяти прошло успешно, нулевой указатель помещается в член `next` вновь созданного элемента, к которому можно обратиться как к `listPtr->next`.

Функция возвращает указатель на новый элемент списка или нулевой указатель, если память не была выделена (для этого необходимо выполнить проверку). Если вы изобразите связанный список и выполните все этапы добавления нового элемента, то вы лучше поймете, как эта функция работает.

Еще одна функция, связанная с динамическим выделением памяти, называется `realloc`. Она может использоваться для сокращения или расширения уже распределенной памяти. Более подробно об этом можно узнать из Приложения Б.

Эта глава завершает обзор возможностей языка программирования C. В главе 18, “Отладка программ”, вы познакомитесь с некоторыми приемами отладки программ.

Отладка программ

В этой главе вы познакомитесь с двумя способами отладки программ. В первом случае для отладки программ используется препроцессор, что позволяет дополнительно вставлять в программу отладочные утверждения. Во втором случае используется интерактивный отладчик. В этой главе вы познакомитесь в популярным отладчиком `gdb`.

Отладка с помощью препроцессора

Как отмечалось в главе 13, “Препроцессор”, при отладке программ весьма удобно использовать условную компиляцию. Препроцессор языка программирования C может использоваться для вставки отладочного кода в разрабатываемую программу. Утверждения `#ifdef` можно использовать для получения необходимого эффекта. Программа из листинга 18.1 (хотя и несколько искусственная), считывает три целых числа и печатает их сумму. Обратите внимание, что, когда идентификатор `DEBUG` установлен, отладочный код, который выполняет вывод в поток `stderr`, компилируется вместе с остальной программой, а когда `DEBUG` не установлен, отладочный код пропускается.

Листинг 18.1. Вставка отладочных утверждений для препроцессора

```
#include <stdio.h>
#define DEBUG
int process (int i, int j, int k)
{
    return i + j + k;
}
int main (void)
{
    int i, j, k, nread;
    nread = scanf ("%d %d %d", &i, &j, &k);
    #ifdef DEBUG
        fprintf (stderr, "Number of integers read = %i\n", nread);
        fprintf (stderr, "i = %i, j = %i, k = %i\n", i, j, k);
    #endif
    printf ("%i\n", process (i, j, k));

    return 0;
}
```

Листинг 18.1. Вывод

```
1 2 3
Number of integers read = 3
i = 1, j = 2, k = 3
6
```

Листинг 18.1. Вывод (Повторно)

```
1 2 e
Number of integers read = 2
i = 1, j = 2, k = 0
3
```

Обратите внимание, что значение для переменной *k* может быть любым, поскольку его значение не устанавливается при вызове функции *scanf* и не инициализируется в программе.

Утверждения

```
#ifdef DEBUG
    fprintf (stderr, "Number of integers read = %i\n", nread);
    fprintf (stderr, "i = %d, j = %d, k = %d\n", i, j, k);
#endif
```

анализируются препроцессором. Если идентификатор *DEBUG* был предварительно использован в утверждении *#define DEBUG*, препроцессор посылает утверждения, которые следуют за директивой *#ifdef DEBUG* в компилятор для компиляции. Если идентификатор *DEBUG* предварительно не был установлен, то эти два утверждения *fprintf* никогда не попадут в компилятор (они будут удалены из программы препроцессором).

Как видите, программа выводит сообщение после того, как она считывает целые числа. При повторном запуске программы вводится символ *e*, что является ошибкой. Если отладочные утверждения подключены, то вы получите сообщение об ошибке. Но если вы отключите отладочные утверждения, для чего необходимо просто убрать утверждение.

```
#define DEBUG
```

то вы не узнаете об ошибке и утверждения *fprintf* не будут компилироваться с остальной программой.

Представьте, что так же легко подключать и отключать отладочные утверждения в программе длиной всего в несколько сот строк, изменяя всего одно утверждение.

Контролировать отладку можно и с командной строки при компиляции программы. Если используется компилятор *gcc*, то команда

```
gcc -D DEBUG debug.c
```

будет компилировать файл *debug.c* установленной переменной *DEBUG*. Это равносильно тому, как будто в программе написано следующее утверждение.

```
#define DEBUG
```

Рассмотрим несколько большую программу. В листинге 18.2 приведена программа, в которой принимаются два аргумента из командной строки. Каждый из них затем преобразовывается в целое число и присваивается переменным *arg1* и *arg2* соответственно. Для преобразования аргументов командной строки в целые числа

используется функция `atoi` из стандартной библиотеки. Эта функция принимает в качестве аргумента строку и возвращает соответствующее целочисленное значение. Функция `atoi` объявлена в заголовочном файле `<stdlib.h>`, который подключается к программе в начале листинга 18.2.

После обработки аргументов в программе вызывается функция `process`, в которую передаются полученные целочисленные значения. Эта функция производит перемножение этих чисел и возвращает результат. Как можно видеть, когда идентификатор `DEBUG` установлен, будут выводиться различные отладочные утверждения, а когда идентификатор `DEBUG` не установлен — будет выведен только результат.

Листинг 18.2. Компиляция с отладочным кодом

```
#include <stdio.h>
#include <stdlib.h>
int process (int i1, int i2)
{
    int val;
    #ifdef DEBUG
        fprintf (stderr, "process (%i, %i)\n", i1, i2);
    #endif
    val = i1 * i2;
    #ifdef DEBUG
        fprintf (stderr, "return %i\n", val);
    #endif
    return val;
}

int main (int argc, char *argv[])
{
    int arg1 = 0, arg2 = 0;
    if (argc > 1)
        arg1 = atoi (argv[1]);
    if (argc == 3)
        arg2 = atoi (argv[2]);
    #ifdef DEBUG
        fprintf (stderr, "processed %i arguments\n", argc - 1);
        fprintf (stderr, "arg1 = %i, arg2 = %i\n", arg1, arg2);
    #endif
    printf ("%i\n", process (arg1, arg2));
    return 0;
}
```

Листинг 18.2. Вывод

```
$ gcc -D DEBUG p18-2.c Compile with DEBUG defined
$ a.out 5 10
processed 2 arguments
arg1 = 5, arg2 = 10
process (5, 10)
return 50
50
```

Листинг 18.2. Вывод (Повторение)

```
$ gcc p18-2.c Compile without DEBUG defined
$ a.out 2 5
10
```

Когда программа готова для распространения, отладочные утверждения могут быть оставлены в исходном коде и они не будут оказывать влияния на исполняемый код, если не устанавливать переменную `DEBUG`. Если в дальнейшем в программе будут обнаружены ошибки, то их будет легче обнаружить с помощью уже написанного отладочного кода, для чего надо программу вновь откомпилировать с установленной переменной `DEBUG`.

Но необходимо отметить, что такой способ довольно неудобен, поскольку с отладочными утверждениями программа становится довольно трудной для понимания. Поэтому необходимо изменить способ использования препроцессора.

Например, можно задать макроопределение, которое будет принимать различное количество аргументов для вывода отладочных сообщений

```
#define DEBUG(fmt, ...) fprintf (stderr, fmt, __VA_ARGS__)
```

и использовать его вместо функции `fprintf` следующим образом.

```
DEBUG ("process (%i, %i)\n", i1, i2);
```

В конечном итоге получим следующее.

```
fprintf (stderr, "process (%i, %i)\n", i1, i2);
```

Макроопределение `DEBUG` можно использовать в программе везде, что позволит придать ей более читабельный вид, как показано в листинге 18.3.

Листинг 18.3. Задание макроопределения `DEBUG`

```
#include <stdio.h>
#include <stdlib.h>
#define DEBUG(fmt, ...) fprintf (stderr, fmt, __VA_ARGS__)
int process (int i1, int i2)
{
    int val;
    DEBUG ("process (%i, %i)\n", i1, i2);
    val = i1 * i2;
    DEBUG ("return %i\n", val);
    return val;
}

int main (int argc, char *argv[])
{
    int arg1 = 0, arg2 = 0;
    if (argc > 1)
        arg1 = atoi (argv[1]);
    if (argc == 3)
        arg2 = atoi (argv[2]);
    DEBUG ("processed %i arguments\n", argc - 1);
    DEBUG ("arg1 = %i, arg2 = %i\n", arg1, arg2);
    printf ("%d\n", process (arg1, arg2));
    return 0;
}
```

Листинг 18.3. Вывод

```
$ gcc pre3.c
$ a.out 8 12
processed 2 arguments
arg1 = 8, arg2 = 12
process (8, 12)
return 96
96
```

Как видим, в таком виде программа более читабельна. Когда отладочная информация уже будет не нужна, просто задайте макроопределение таким образом, чтобы оно ничего не выполняло.

```
#define DEBUG(fmt, ...)
```

Такое задание макроопределения приведет к тому, что препроцессор ничем не заменит вызов макроопределения `DEBUG`, т.е. все вызовы макроопределения `DEBUG` будут просто убраны.

Можно расширить использование макроопределения `DEBUG` таким образом, чтобы можно было производить контроль как во время компиляции, так и во время выполнения программы. Для этого задайте глобальную переменную `Debug`, которая будет определять уровень отладки. Все отладочные утверждения, меньшие или равные этому уровню, будут производить вывод. Макроопределение `DEBUG` теперь должно принимать два аргумента, первый из которых будет представлять уровень отладки.

```
DEBUG (1, "processed data\n");
DEBUG (3, "number of elements = %i\n", nelems)
```

Если уровень отладки будет установлен в 1 или 2, то только первое утверждение `DEBUG` будет производить вывод, а если уровень отладки будет остановлен в 3, то вывод будут производить оба утверждения. Уровень отладки можно будет устанавливать с помощью опций командной строки во время выполнения, как показано ниже.

```
a.out -d1    Set debugging level to 1
a.out -d3    Set debugging level to 3
```

Задать макроопределение `DEBUG` можно следующим образом.

```
#define DEBUG(level, fmt, ...) \
if (Debug >= level) \
fprintf (stderr, fmt, __VA_ARGS__)
```

Поэтому выражение

```
DEBUG (3, "number of elements = %i\n", nelems);
```

будет преобразовано в следующее.

```
if (Debug >= 3)
fprintf (stderr, "number of elements = %i\n", nelems);
```

Как и ранее, если ничего не задать для макроопределения, то это приведет к тому, что все соответствующие утверждения будут убраны. Следующее утверждение использует все указанные возможности и позволяет контролировать макроопределения `DEBUG` во время компиляции.

```

#ifdef DEBON
    # define DEBUG(level, fmt, ...) \
    if (Debug >= level) \
        fprintf(stderr, fmt, __VA_ARGS__)
#else
    # define DEBUG(level, fmt, ...)
#endif

```

Когда компилируется программа, содержащая предыдущее утверждение (которое для удобства может располагаться в заголовочном файле, который подключается к программе), то установка переменной DEBON не является обязательной. Если компилировать программу с помощью команды

```
$ gcc prog.c
```

то это означает, что при компиляции не будет производиться вывод отладочной информации, поскольку вместо макроопределения DEBUG будет подставлено пустое выражение. Но если компилировать программу с помощью команды

```
$ gcc -D DEBON prog.c
```

то это приведет к вызову функций `fprintf` в зависимости от заданного уровня отладки.

Если компиляция производилась с включенной отладкой, во время запуска на выполнение можно задать уровень отладки. Это можно сделать в командной строке, используя соответствующие опции.

```
$ a.out -d3
```

В данной команде уровень отладки установлен в 3. Возможно, вы будете использовать этот уровень отладки в вашей программе и сохраните этот уровень в переменной (возможно, глобальной) с именем `Debug`. В этом случае макроопределение `DEBUG` будет вызывать функции `fprintf` при задании уровня отладки 3 или больше.

Обратите внимание, что если установить уровень отладки в 0 (`a.out -d0`), то никакая отладочная информация выводиться не будет, даже если отладочный код остается в программе.

Отметим, что отладочный код может быть легко подключен или отключен. Если отладочный код подключен, то можно задавать различные уровни отладки и производить вывод отладочной информации необходимого объема.

Отладчик gdb

Отладчик `gdb` является мощным интерактивным отладчиком, который часто используется при отладке программ, компилируемых с помощью компилятора `gcc`, распространяемого по лицензии GNU. Он позволяет запускать программу на выполнение, останавливаться в заданных местах, отображать переменные и задавать им значения. После чего он может продолжать выполнение. Это позволяет производить трассировку программы и последовательно выполнять строку за строкой.

Отладчик `gdb` может контролировать вывод предупреждающих сообщений системы, которые возникают в результате аварийного события, например деления на ноль или выхода за пределы массива. В результате будет создан файл, в котором будет содержаться временный снимок памяти процесса (дамп памяти) на момент аварийной ситуации. Хотя можно сконфигурировать систему так, что такой файл выводиться не будет, т.к. его размеры довольно большие.

Если программа компилируется с помощью компилятора gcc, то необходимо использовать опцию `-g` для того, чтобы можно было воспользоваться всеми возможностями отладчика gdb. Опция `-g` заставляет компилятор языка C добавлять дополнительную информацию в выходной файл, включая переменные и структурные типы, исходные имена файлов и утверждения языка для распределения кода. В программе из листинга 18.4 приводится пример обращения к массиву за его пределами.

Листинг 18.4. Простая программа для работы с gdb

```
#include <stdio.h>
int main (void)
{
    const int data[5] = {1, 2, 3, 4, 5};
    int i, sum;
    for (i = 0; i >= 0; ++i)
        sum += data[i];
    printf ("sum = %i\n", sum);
    return 0;
}
```

Ниже показано, что произойдет, когда программа будет запущена на компьютере с операционной системой Mac OS X в режиме терминального окна (на других системах сообщения могут быть другими).

```
$ a.out
Segmentation fault
```

Используем отладчик gdb для поиска ошибки.

Сначала убедимся, что программа была скомпилирована с опцией `-g`. Затем запустим отладчик gdb с файлом `a.out`. В результате работы отладчика будут выведены следующие строки сообщения.

```
$ gcc -g p18.4.c Перекомпилировать с отладочной информацией для gdb
$ gdb a.out Запуск gdb для исполняемого файла
GNU gdb 5.3-20030128 (Apple version gdb-309) (Thu Dec 4 15:41:30 GMT
2003)
Copyright 2003 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and
you are
welcome to change it and/or distribute copies of it under certain
conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB. Type "show warranty" for
details.
This GDB was configured as "powerpc-apple-darwin".
Reading symbols for shared libraries .. done
```

Когда отладчик будет готов принимать команды, он отобразит запрос в виде (gdb). В нашем примере вы можете просто запустить программу на выполнение, введя команду `run`. Это приведет к тому, что программа будет под управлением gdb выполняться до тех пор, пока не произойдет аварийная ситуация.

```
(gdb) run
Starting program: /Users/stevekochan/MySrc/c/a.out
```

```

Reading symbols for shared libraries . done
Program received signal EXC_BAD_ACCESS, Could not access memory.
0x00001d7c in main () at pl8-4.c:9
9 sum += data[i];
(gdb)

```

Как видно из полученного сообщения, отладчик зафиксировал аварийную ситуацию, но программа по-прежнему остается под управлением отладчика gdb. Это довольно удобно, поскольку вы можете видеть, что произошло непосредственно перед аварийной ситуацией, а также видеть значения переменных в момент ошибки.

Из выведенного сообщения можно понять, что программа сделала попытку обратиться к недоступной для нее в данный момент памяти в строке 9. Само содержимое строки программы также отображается в сообщении. Для получения более развернутого фрагмента программы можно использовать команду `list`, после чего будет отображено 10 строк исходного кода, 5 строк до нужной строки и 4 строки после нее.

```

(gdb) list 9
4 {
5 const int data[5] = {1, 2, 3, 4, 5};
6 int i, sum;
7
8 for (i = 0; i <= 4; ++i)
9 sum += data[i];
10
11 printf ("sum = %i\n", sum);
12
13 return 0;
(gdb)

```

Также можно просмотреть и переменные с помощью команды `print`. Введите эту команду с именем переменной `sum` и вы увидите ее значение в момент возникновения ошибки.

```

(gdb) print sum
$1 = -1089203864

```

Очевидно, что это значение совсем не соответствует тому, что вы ожидали (у вас может получиться другое значение). Обозначение `%n` используется отладчиком для того, чтобы зафиксировать отображаемое значение, к которому теперь можно легко обратиться в дальнейшем.

Посмотрим, какое значение будет иметь индекс `i` в момент ошибки.

```

(gdb) print i
$2 = 232

```

Ого! Это совсем не то, что ожидалось. В программе используется только пять элементов массива, а в момент ошибки производится обращение к 232-му элементу. В нашем случае могут получиться другие цифры, но, в любом случае, это будут ошибочные значения.

Перед тем как выйти из отладчика, посмотрим и другие переменные. Убедимся, как удобно работать с gdb для просмотра переменных типа массив или структура.

```

(gdb) print data      Отображает все элементы массива.
$3 = {1, 2, 3, 4, 5}

```

```
(gdb) print data[0]    Отображает значение первого элемента.  
$4 = 1
```

Пример отображения членов структуры будет рассмотрен несколько позже. Для того чтобы завершить работу с отладчиком gdb, необходимо ввести команду quit.

```
(gdb) quit  
The program is running. Exit anyway? (y or n) y  
$
```

Даже если программа имеет ошибки, она, если говорить техническим языком, все еще активна в отладчике gdb. Ошибка приводит к тому, что выполнение программы приостанавливается, но не прекращается. Именно по этой причине отладчик требует подтверждения на закрытие программы.

Работа с переменными

Отладчик gdb включает две базовые команды, которые позволяют работать с переменными в программе. С одной из них вы уже познакомились – это print. Другая команда позволяет установить значение для переменной. Это команда set var. С командой set можно использовать довольно много опций, среди которых опция var является именно той, с помощью которой можно присваивать переменным значения.

```
(gdb) set var i=5  
(gdb) print i  
$1 = 5  
(gdb) set var i=i*2    Можно писать любое корректное выражение.  
(gdb) print i  
$2 = 10  
(gdb) set var i=$1+20  Можно использовать "нумерованные переменные".  
(gdb) print i  
$3 = 25
```

Переменная должна быть доступна в текущей функции и процесс должен быть активным, т.е. запущенный отладчик использует идею текущей строки (подобно текстовому редактору), текущего файла (исходный файл с программой) и текущей функции. Если отладчик gdb запускается без исходного файла, текущей функцией становится функция main, текущим файлом становится тот, что содержит функцию main, а текущей строкой становится первая исполняемая строка функции main. В противном случае устанавливаются текущая строка, текущий файл и функция, которые были при завершении программы.

Если переменной с заданным именем не существует, то отладчик определяет это имя среди внешних переменных. В предыдущем примере функцией, в которой произошло аварийное завершение, была функция main, а переменная i являлась локальной для функции main.

Имя функции может быть указано как часть имени переменной.

```
function::variable
```

Поэтому можно использовать одинаковые имена переменных для ссылок на переменные из различных функций.

```
(gdb) print main::i    Отображение содержимого переменной i
                        из функции main.
$4 = 25
(gdb) set var main::i=0  Установка значения переменной i из функции
                        main.
```

Обратите внимание, что попытка установить значение для неактивной переменной (например, если функция не является в текущий момент активной или находится в режиме ожидания выполнения другой функции для продолжения работы), то произойдет ошибка и будет выведено следующее сообщение.

```
No symbol "var" in current context.
```

На глобальные переменные можно непосредственно ссылаться как 'file':var.

При этом отладчик обращается к внешней переменной в файле file и игнорирует локальные переменные с таким же именем в текущей функции.

К членам структур и объединений можно обращаться с помощью стандартного синтаксиса языка C. Если указатель datePtr указывает на тип date structure, то выражение

```
print datePtr->year
```

приведет к выводу значения члена year из структуры, на которую указывает datePtr.

Ссылка на структуру или объединение без указания члена приведет к выводу всего содержимого структуры или объединения.

Вы можете заставить отладчик gdb отображать переменные в различных форматах, например, в шестнадцатеричном, сопровождая команду print обратной чертой с символом, задающим необходимый формат. Многие команды отладчика могут быть сокращены до одной буквы. В следующем примере для команды print будет использоваться аббревиатура p.

```
(gdb) set var i=35      Задать для i значение 35.
(gdb) p /x i           Отобразить i в шестнадцатеричном формате.
$1 = 0x23
```

Отображение исходного файла

Отладчик gdb имеет несколько команд, с помощью которых можно получить доступ к исходному файлу. Это позволяет отлаживать программы без обращения к листингу или открытия исходного файла в отдельном окне.

Как уже упоминалось, отладчик gdb использует понятия текущего файла и текущей строки. Вы уже познакомились с тем, как можно отобразить строки, примыкающие к текущей строке, с помощью команды list, которую можно заменить аббревиатурой l. При последовательном вводе команды list (при этом можно только нажимать клавиши <Enter> или <Return>, что значительно удобнее) будут отображаться следующие 10 строк исходного текста. Значение для 10 строк используется по умолчанию и может быть установлено другим с помощью команды listsize.

Если необходимо отобразить некоторый диапазон строк, то можно задать номера начальной и конечной строк, разделенных точкой с запятой, как показано ниже.

```
(gdb) list 10,15      Строки от 10 до 15
```

Все строки, составляющие функцию, также можно вывести с помощью команды `list`, указав в качестве аргумента имя функции.

```
(gdb) list foo      Отображает строки, составляющие функцию foo
```

Если функция находится в другом исходном файле, то отладчик `gdb` автоматически переключается на этот файл. Имя текущего файла можно получить, если ввести команду `info source`.

Использование в команде `list` символа `+` после команды приводит к тому, что будут выведены последующие 10 строк кода из текущего файла, т.е. это тоже самое, что просто ввести команду `list`. Использование в команде `list` символа `-` после команды приводит к тому, что будут выведены предыдущие 10 строк кода из текущего файла. Символы `+` и `-` могут сопровождаться числом, которое задает относительное смещение, которое должно быть прибавлено или вычтено из номера текущей строки.

Контроль выполнения программы

Отображение строк из файла не изменяет последовательности выполнения команд. Для изменения последовательности выполнения необходимо использовать другие команды. Вы уже познакомились с двумя командами отладчика `gdb`, которые контролируют выполнение программы, — это команда `run`, которая запускает программу на выполнение с самого начала, и команда `quit`, которая прекращает выполнение текущей программы.

Команда `run` может сопровождаться аргументами командной строки и командами перенаправления, и все это отладчик `gdb` корректно обработает. Последовательное использование команды `run` без аргументов приведет к тому, что будут использоваться предыдущие аргументы и перенаправления. Текущие аргументы можно отобразить с помощью команды `show args`.

Вставка точек останова

Команда `break` может использоваться для задания точек останова в программе. Точкой останова, как можно понять из названия команды, считается та точка или строка программы, при достижении которой в процессе выполнения будет произведен временный “останов” программы, или пауза. Выполнение программы будет прервано для того, чтобы вы могли проанализировать содержимое переменных и определить текущее состояние вычислений.

Точку останова задать легко, для чего необходимо только указать номер строки программы в команде. Если вы задали номер строки, но не имя функции или файла, то эта строка будет установлена в текущем файле. Если задано имя функции, то останов произойдет в первой исполняемой строке указанной функции.

```
(gdb) break 12          Останов будет сделан в строке 12  
Breakpoint 1 at 0x1da4: file mod1.c, line 12.  
(gdb) break main       Останов будет сделан в начале функции main  
Breakpoint 2 at 0x1d6c: file mod1.c, line 3.  
(gdb) break mod2.c:foo  Останов будет сделан в функции foo из файла  
                             mod2.c  
Breakpoint 3 at 0x1dd8: file mod2.c, line 4.
```

При достижении точки останова, отладчик `gdb` приостанавливает выполнение программы, передает управление вам и выделяет строку, где произошла остановка выполнения программы. В этот момент вы можете делать что пожелаете: можно

отобразить набор переменных, задать новые или сбросить уже установленные точки останова и т.д. Для продолжения выполнения программы, необходимо ввести команду `continue`, которую можно заменить аббревиатурой `c`.

Пошаговое выполнение

Другой полезной командой для контроля выполнения программы является команда `step`, которую можно сократить до аббревиатуры `s`. Эта команда выполняет один шаг программы, что означает выполнение только одной строки программы на языке C при каждом вводе команды `step`. Если за командой `step` указать число, тогда будет выполнено несколько строк кода в соответствии с заданным числом. При этом обратите внимание, что на одной строке может находиться несколько утверждений. Отладчик `gdb` ориентирован на строки, и поэтому будут выполнены все утверждения, находящиеся на строке. Вы можете производить пошаговое выполнение в любое время, тогда как команда `continue` может выполняться только при соответствующих условиях.

Если в утверждении содержится вызов функции, то при вводе команды `step` отладчик перейдет к выполнению команд функции, при условии, что это не функция из стандартной библиотеки, вход в которую обычно закрыт. Если вы используете вместо команды `step` команду `next`, отладчик произведет вызов функции, но не будет последовательно выполнять утверждения функции (заходить в функцию), а сразу получит результат выполнения функции. Изучите программу из листинга 18.5, в которой используются некоторые полезные возможности отладчика `gdb`.

Листинг 18.5. Работа с отладчиком `gdb`

```
#include <stdio.h>
#include <stdlib.h>
struct date
{
    int month;
    int day;
    int year;
};
struct date foo (struct date x)
{
    ++x.day;
    return x;
}

int main (void)
{
    struct date today = {10, 11, 2004};
    int array[5] = {1, 2, 3, 4, 5};
    struct date *newdate, foo ();
    char *string = "test string";
    int i = 3;

    newdate = (struct date *) malloc (sizeof (struct date));
    newdate->month = 11;
    newdate->day = 15;
    newdate->year = 2004;
    today = foo (today);
    free (newdate);

    return 0;
}
```

В простом примере, показанном в листинге 18.6, вывод может отличаться в зависимости от системы, на которой будет запущен отладчик gdb, причем это зависит и от версии отладчика.

Листинг 18.6. Использование gdb

```
$ gcc -g p18-5.c
$ gdb a.out
GNU gdb 5.3-20030128 (Apple version gdb-309) (Thu Dec 4 15:41:30 GMT
2003)
Copyright 2003 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and
you are
welcome to change it and/or distribute copies of it under certain
conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB. Type "show warranty" for
details.
This GDB was configured as "powerpc-apple-darwin".
Reading symbols for shared libraries .. done
(gdb) list main
14
15 return x;
16 }
17
18 int main (void)
19 {
20 struct date today = {10, 11, 2004};
21 int array[5] = {1, 2, 3, 4, 5};
22 struct date *newdate, foo ();
23 char *string = "test string";
(gdb) break main          Задание точки останова.
Breakpoint 1 at 0x1ce8: file p18-5.c, line 20.
(gdb) run                  Запуск программы на выполнение.
Starting program: /Users/stevekochan/MySrc/c/a.out
Reading symbols for shared libraries .. done
Breakpoint 1, main () at p18-5.c:20
20 struct date today = {10, 11, 2004};
(gdb) step                 Выполнить строку 20.
21 int array[5] = {1, 2, 3, 4, 5};
(gdb) print today
$1 = {
month = 10,
day = 11,
year = 2004
}
(gdb) print array          Этот массив еще не инициализирован.
$2 = {-1881069176, -1880816132, -1880815740, -1880816132, -1880846287}
(gdb) step                 Выполнить последующую строку.
23 char *string = "test string";
(gdb) print array          Сейчас можно.
$3 = {1, 2, 3, 4, 5}       Это правильно.
(gdb) list 23,28
23 char *string = "test string";
24 int i = 3;
```

```

25
26 newdate = (struct date *) malloc (sizeof (struct date));
27 newdate->month = 11;
28 newdate->day = 15;
(gdb) step 5                                Выполнить 5 строк.
29 newdate->year = 2004;
(gdb) print string
$4 = 0x1fd4 "test string"
(gdb) print string[1]
$5 = 101 'e'
(gdb) print array[i]                        Программно i установлен в 3.
$6 = 3
(gdb) print newdate                          Это указатель на переменную.
$7 = (struct date *) 0x100140
(gdb) print newdate->month
$8 = 11
(gdb) print newdate->day + i                  Допустимое выражение.
$9 = 18
(gdb) print $7                                Доступ к предыдущему значению.
$10 = (struct date *) 0x100140
(gdb) info locals                            Показать значения всех локальных переменных.
today = {
  month = 10,
  day = 11,
  year = 2004
}
array = {1, 2, 3, 4, 5}
newdate = (struct date *) 0x100140
string = 0x1fd4 "test string"
i = 3
(gdb) break foo                                Точка останова в начале функции foo.
Breakpoint 2 at 0x1c98: file p18-5.c, line 13.
(gdb) continue                                Продолжить выполнение.
Continuing.
Breakpoint 2, foo (x={month = 10, day = 11, year = 2004}) at
p18-5.c:13
13 ++x.day; 0x8e in foo:25: {
(gdb) print today                            Отобразить значение переменной today
No symbol "today" in current context
(gdb) print main::today                       Отобразить значение переменной today из main
$11 = {
  month = 10,
  day = 11,
  year = 2004
}
(gdb) step
15 return x;
(gdb) print x.day
$12 = 12
(gdb) continue
Continuing.
Program exited normally.
(gdb)

```

Обратите внимание на одну из особенностей отладчика gdb: после достижения точки останова или после выполнения одного шага, он выводит строку, которая будет выполняться следующей после возобновления выполнения программы, а не последнее выполненное утверждение. Именно потому переменная `array` не была инициализирована при первом выводе на экран.

Выполнение одного шага приводит к тому, что переменная инициализируется. Также обратите внимание на то, что инициализация переменной рассматривается как выполняемая строка, хотя при этом компилятор не создает выполняемой код.

Список и удаление точек останова

Однажды установленные, точки останова остаются в программе до тех пор, пока производится работа с отладчиком или пока вы не уберете их принудительно. Вы можете просмотреть все точки останова, которые были заданы, с помощью команды `info break`, как показано ниже.

```
(gdb) info break
Num Type Disp Enb Address What
1 breakpoint keep y 0x00001c9c in main at p18-5.c:20
2 breakpoint keep y 0x00001c4c in foo at p18-5.c:13
```

Вы можете удалить точку останова для отдельной строки с помощью команды `clear`, за которой следует номер строки. Для функции точку останова можно удалить тоже с помощью команды `clear`, но вместо номера строки необходимо вставить имя функции.

```
(gdb) clear 20      Удаление точки останова для строки 20
Deleted breakpoint 1
(gdb) info break
Num Type Disp Enb Address What
2 breakpoint keep y 0x00001c4c in foo at p18-5.c:13
(gdb) clear foo     Удаление точки останова для функции foo.
Deleted breakpoint 2
(gdb) info break
No breakpoints or watchpoints.
(gdb)
```

Трассировка стека

Иногда необходимо точно знать, какое место занимает функция в иерархии вызовов при останове программы. Такая информация полезна при анализе взаимоотношений функций в программе. Для получения этой информации можно просмотреть стек вызовов с помощью команды `backtrace`, которая также может быть заменена аббревиатурой `bt`. Ниже приведен пример использования стека вызовов.

```
(gdb) break foo
Breakpoint 1 at 0x1c4c: file p18-5.c, line 13.
(gdb) run
Starting program: /Users/stevekochan/MySrc/c/a.out

Reading symbols for shared libraries . done
Breakpoint 1, foo (x={month = 10, day = 11, year = 2004}) at
p18-5.c:13
```

```

13 ++x.day;
(gdb) bt                                Вывести трассировку стека.
#0 foo (x={month = 10, day = 11, year = 2004}) at p18-5.c:13
#1 0x00001d48 in main () at p18-5.c:31
(gdb)

```

Если останов происходит при входе в функцию `foo`, вызывается команда `backtrace`. Из вывода, сделанного для этой команды, можно видеть, что в стеке вызовов находятся две функции: `foo` и `main`. При этом выводятся и аргументы функций. Различные команды, такие как `up`, `down`, `frame`, `info args`, которые здесь подробно не рассматриваются, позволяют исследовать стек и получить подробную информацию об аргументах, переданных в отдельную функцию, или просмотреть локальные переменные.

Вызов функций и установка массивов и структур

В отладчике `gdb` вы можете легко вызвать функцию.

```

(gdb) print foo(*newdate)              Вызов функции foo со структурой,
                                         на которую указывает newdate.

$13 = {
month = 11,
day = 16,
year = 2004
}
(gdb)

```

Функция `foo` определена в листинге 18.5. Вы также можете присвоить значения массиву или структуре, перечисляя значения внутри фигурных скобок.

```

(gdb) print array
$14 = {1, 2, 3, 4, 5}
(gdb) set var array = {100, 200}
(gdb) print array
$15 = {100, 200, 0, 0}  Неприсвоенные элементы устанавливаются
                        в нуль.

(gdb) print today
$16 = {
month = 10,
day = 11,
year = 2004
}
(gdb) set var today={8, 8, 2004}
(gdb) print today
$17 = {
month = 8,
day = 8,
year = 2004
}
(gdb)

```

Получение справки с помощью gdb

Вы можете использовать встроенную команду `help` для получения справки о различных командах или типах команд (в отладчике `gdb` называемых классами). Команда `help`, введенная без аргументов, выводит все доступные классы.

```
(gdb) help
```

```
List of classes of commands:
```

```
aliases -- Aliases of other commands
```

```
breakpoints -- Making program stop at certain points
```

```
data -- Examining data
```

```
files -- Specifying and examining files
```

```
internals -- Maintenance commands
```

```
obscure -- Obscure features
```

```
running -- Running the program
```

```
stack -- Examining the stack
```

```
status -- Status inquiries
```

```
support -- Support facilities
```

```
tracepoints -- Tracing of program execution without stopping the  
program
```

```
user-defined -- User-defined commands
```

```
Type "help" followed by a class name for a list of commands in that  
class.
```

```
Type "help" followed by command name for full documentation.
```

```
Command name abbreviations are allowed if unambiguous.
```

Но можно ввести команду `help` с одним из перечисленных выше классов.

```
(gdb) help breakpoints
```

```
Making program stop at certain points.
```

```
List of commands:
```

```
awatch -- Set a watchpoint for an expression
```

```
break -- Set breakpoint at specified line or function
```

```
catch -- Set catchpoints to catch events
```

```
clear -- Clear breakpoint at specified line or function
```

```
commands -- Set commands to be executed when a breakpoint is hit
```

```
condition -- Specify breakpoint number N to break only if COND is  
true
```

```
delete -- Delete some breakpoints or auto-display expressions
```

```
disable -- Disable some breakpoints
```

```
enable -- Enable some breakpoints
```

```
future-break -- Set breakpoint at expression
```

```
hbreak -- Set a hardware assisted breakpoint
```

```
ignore -- Set ignore-count of breakpoint number N to COUNT
```

```
rbreak -- Set a breakpoint for all functions matching REGEXP
```

```
rwatch -- Set a read watchpoint for an expression
```

```
save-breakpoints -- Save current breakpoint definitions as a script
```

```
set exception-catch-type-regexp -
```

```
Set a regexp to match against the exception type of a caughtobject
```

```
set exception-throw-type-regexp -
```

```
Set a regexp to match against the exception type of a thrownobject
```

```
show exception-catch-type-regexp -
```

Show a regexp to match against the exception type of a caught object
 show exception-throw-type-regexp -
 Show a regexp to match against the exception type of a thrown object
 tbreak -- Set a temporary breakpoint
 tcatch -- Set temporary catchpoints to catch events
 thbreak -- Set a temporary hardware assisted breakpoint
 watch -- Set a watchpoint for an expression
 Type "help" followed by command name for full documentation.
 Command name abbreviations are allowed if unambiguous.
 (gdb)

Аналогично, в качестве аргумента можно использовать одну из перечисленных выше команд.

```
(gdb) help break
Set breakpoint at specified line or function.
Argument may be line number, function name, or "*" and an address.
If line number is specified, break at start of code for that line.
If function is specified, break at start of code for that function.
If an address is specified, break at that exact address.
With no arg, uses current execution address of selected stack frame.
This is useful for breaking on return to a stack frame.
Multiple breakpoints at one place are permitted, and useful if
conditional.
break ... if <cond> sets condition <cond> on the breakpoint as it is
created.
Do "help breakpoints" for info on other commands dealing with
breakpoints.
(gdb)
```

Как видим, вы можете получить достаточное количество информации находясь непосредственно в отладчике gdb. Старайтесь не упустить этот шанс!

Дополнительные возможности

Из-за ограничений по объему невозможно описать все многочисленные возможности отладчика gdb. Представим лишь некоторые из них.

- Установка временных точек останова, которые будут автоматически удаляться при их достижении.
- Активизация или деактивизация точек останова без их удаления.
- Выведение дампа памяти в необходимом формате.
- Установка наблюдателей, что позволяет производить останов выполняющейся программы при изменении указанного значения, например, при изменении значения переменной.
- Возможность задавать список переменных, которые будут отображаться при остановке программы.
- Возможность задавать собственные "оперативные переменные" по именам.

В табл. 18.1 перечислены команды отладчика gdb, о которых говорилось в этой главе. Выделенные жирным шрифтом буквы показывают соответствующую аббревиатуру для команды.

Таблица 18.1. Общие команды отладчика gdb

Команда	Значение
<i>Исходный файл</i>	
list [n]	Отображает строки, примыкающие к строке n, или следующие 10 строк, если n не указано
list m,n	Отображает строки с номерами от m до n
list +[n]	Отображает строки, примыкающие к строке n или последующие 10 строк, если n не указано
list -[n]	Отображает строки, примыкающие к строке n, или предыдущие 10 строк, если n не указано
list func	Отображает строки из функции func
listsize n	Задаёт количество строк для отображения по команде list
info source	Отображает имя текущего файла
<i>Переменные и выражения</i>	
print /fmt expr	Выводит expr в соответствии с форматом fmt, который может соответствовать: d (десятичный), u (без знака), o (восьмеричный), x (шестнадцатеричный), c (символьный), f (с плавающей запятой), t (двоичный) или a (адрес)
info locals	Отображает значения локальных переменных для текущей функции
set var; var=expr	Присваивает переменной var значение expr
<i>Точки останова</i>	
break n	Устанавливает точку останова в строке n
break func	Устанавливает точку останова в начале функции func
info break	Показывает все точки останова
clear [n]	Удаляет точку останова из строки n
clear func	Удаляет точку останова из функции func
<i>Выполнение программы</i>	
run [args] [<file> [>file]	Выполнить программу с самого начала
continue	Продолжить выполнение программы
step [n]	Выполнить следующую строку программы или n строк
next [n]	Выполняет следующую строку программы или следующие n строк программы без захода в функции
quit	Окончание работы gdb
<i>Справка</i>	
help [cmd]	Отображает классы команд или справки по отдельным командам cmd
help [class]	или классам

Объектно-ориентированное программирование

Поскольку объектно-ориентированное программирование (ООП) в настоящее время широко распространено и поскольку такие популярные языки программирования, как C++, C#, Java и Objective-C, использующие принципы ООП, основаны на языке программирования C, в данной книге представлено краткое описание этой технологии.

Что такое объект?

Объект — это сущность. Думая об объектно-ориентированном программировании, вы думаете о наборе сущностей и о том, что можно сделать с этими сущностями. Это не то же самое, что программирование на языке C, который формально называется процедурным языком программирования. В отношении языка программирования C, вы сначала думаете о том, что вы хотите сделать, или даже пишете несколько функций для решения поставленной задачи, и только после этого начинаете компоновать фрагменты, которые можно рассматривать как объекты.

В качестве примера из повседневной жизни рассмотрим ваш автомобиль. Очевидно, этот автомобиль можно рассматривать как объект, который принадлежит вам. У вас больше нет другого автомобиля, вы имеете свой отдельный автомобиль, который произведен на фабрике, возможно, в Детройте, а возможно, в Японии или еще где-то. Ваш автомобиль имеет идентификационный номер, который уникально его идентифицирует.

Выражаясь терминами объектно-ориентированного программирования, ваш автомобиль является экземпляром автомобиля. И продолжая применять термины объектно-ориентированного программирования, можно сказать, что слово “автомобиль” является именем класса, от которого были произведены экземпляры автомобиля. Таким образом, когда на заводе выпускается очередной автомобиль, можно сказать, что создается новый экземпляр класса автомобиль.

На каждый экземпляр класса автомобиль можно ссылаться как на *объект*. Ваш автомобиль может быть серебристого цвета с черной внутренней отделкой, это может быть автомобиль с откидным верхом или седан и т.д. При этом вы можете выполнять некоторые действия в вашем автомобиле. Вы можете на нем ездить, можете его заправлять бензином или чистить и мыть его и т.д. (табл. 19.1).

Таблица 19.1. Действия с объектом

Объект	Что с ним можно делать
Ваш автомобиль	Водить; Заправлять бензином; Мыть; Обслуживать

Все действия, которые приведены в табл. 19.1, могут производиться не только с нашим автомобилем, но и со многими другими. Например, ваша сестра тоже имеет автомобиль, и тоже водит его, моет и заправляет бензином.

Экземпляры и методы

Уникальное воспроизведение класса — это экземпляр. Действия, которые вы выполняете с экземпляром, называются *методами*. В некоторых классах методы могут использоваться только с экземплярами класса, а в других — в самом классе. Например, мытье автомобиля может быть выполнено только с отдельным экземпляром автомобиля (да и все остальные методы, описанные в таблице 19.1). Но, выпуская различные типы автомобилей, производитель использует методы класса, которые применяются ко всем автомобилям.

В языке программирования C++ вызов метода экземпляра производится следующим образом.

```
Instance.method ();
```

В языке программирования C# вызов метода экземпляра производится аналогичным образом.

```
Instance.method ();
```

В языке программирования Objective-C вызов метода экземпляра производится так, как показано ниже.

```
[Instance method]
```

Теперь вернемся к списку действий, приведенному выше, и запишем выражения для действий в новом синтаксисе. Предположим, что объект `yourCar` является экземпляром класса `Car`. В табл. 19.2 показаны выражения, которые могут использоваться в трех объектно-ориентированных языках.

Таблица 19.2. Выражения в объектно-ориентированных языках

C++	C#	Objective-C	Действие
<code>yourCar.drive()</code>	<code>yourCar.drive()</code>	<code>[yourCar drive]</code>	Водить автомобиль
<code>yourCar.getGas()</code>	<code>yourCar.getGas()</code>	<code>[yourCar getGas]</code>	Заправлять бензином
<code>yourCar.wash()</code>	<code>yourCar.wash()</code>	<code>[yourCar wash]</code>	Мыть
<code>yourCar.service()</code>	<code>yourCar.service()</code>	<code>[yourCar service]</code>	Обслуживать

Если ваша сестра тоже имеет автомобиль, который можно назвать, например, `suesCar`, то тогда вызов определенного метода для ее автомобиля будет выглядеть следующим образом.

```
suesCar.drive()  
suesCar.drive()  
[suesCar drive]
```

В этом и заключается основная идея объектно-ориентированного программирования — в использовании одного и того же метода с различными объектами.

Еще одной ключевой концепцией объектно-ориентированного программирования является *полиморфизм*. Например, если у вас есть класс `Boat` (Лодка) и вы имеете экземпляр этого класса `myBoat`, то вы также можете использовать для него вышеперечисленные методы, т.е. можно записать на языке C++ следующее.

```
myBoat.service()  
myBoat.wash()
```

Главное, что вы можете использовать метод “обслуживание” для класса `Boat`, хотя это не то же самое, что обслуживание автомобиля. Именно это и составляет суть полиморфизма.

Необходимо понять важное различие между объектно-ориентированными языками программирования и языком программирования C. В первом случае вы работаете с объектами, такими как автомобиль или лодка. Во втором случае вы обычно работаете с функциями (или процедурами). Языки, подобные языку программирования C, называются процедурными языками программирования. Вы можете написать функцию с именем `service` и затем внутри функции реализовать код, который выполняет действия по обслуживанию автомобиля, лодки или мотоцикла. Если вы захотите добавить новый тип средства передвижения, вам нужно будет модифицировать все функции, которые реализуют действия, производимые с данными средствами.

В случае с языками ООП, вы только определите новый класс и добавите новые методы в этот класс. Вам не надо будет беспокоиться о других классах, т.к. они независимы от данного класса и вам не придется модифицировать большое количество строк кода, к которым, возможно, вы даже не будете иметь возможности обратиться.

Классами, с которыми производится работа в языках ООП, скорее всего, будут не автомобили и лодки. Вероятнее всего, это будут объекты, подобные окнам, прямоугольникам, буферам обмена и т.д. Выражения, которые вы будете писать, будут подобны следующим выражениям на языке C#.

<code>myWindow.erase()</code>	Удалить окно
<code>myRect.getArea()</code>	Рассчитать площадь
<code>прямоугольника</code>	
<code>userText.spellCheck()</code>	Проверить текст
<code>deskCalculator.setAccumulator(0.0)</code>	Очистить аккумулятор
<code>favoritePlaylist.showSongs()</code>	Показать названия песен
	из основного файла

Программы для работы с дробями

Предположим, вам необходимо написать программу для работы с дробями. Возможно, необходимо будет производить такие действия, как сложение, вычитание, перемножение, деление и др. Вы можете задать структуру, содержащую дробь, и затем разработать функции, производящие над ней действия. Основные решения для работы с дробями могут быть подобны тем, что показаны в программе из листинга 19.1. В этой программе задаются числитель и знаменатель дроби и затем отображаются значения дробного числа.

Листинг 19.1. Работа с дробями на языке C

```
// Простая программа для работы с дробями.
#include <stdio.h>
typedef struct
{
    int numerator;
    int denominator;
} Fraction;

int main (void)
{
    Fraction myFract;
    myFract.numerator = 1;
    myFract.denominator = 3;
    printf ("The fraction is %i/%i\n", myFract.numerator, myFract.
denominator);
    return 0;
}
```

Листинг 19.1. Вывод

The fraction is 1/3

В следующих трех разделах показано, как можно реализовать работу с дробями на языках Objective-C, C++ и C# соответственно. Все утверждения о концепции ООП, представленные в программе из листинга 19.2, применяются и в дальнейшем, поэтому вы должны читать эти разделы последовательно.

Определение класса для работы с дробями на языке Objective-C

Язык Objective-C был придуман Бредом Коксом в начале 1980-х годов. Язык базировался на языке SmallTalk-80 и был лицензирован фирмой NeXT Software в 1988 году. Когда фирма Apple приобрела фирму NeXT в 1988 году, она использовала наработки этой фирмы для создания операционной системы Mac OS X. Большинство программных фрагментов, используемых в настоящее время в Mac OS X, написаны на языке Objective-C.

В программе из листинга 19.2 показано, как можно определить и использовать класс Fraction на языке Objective-C.

Листинг 19.2. Работа с дробями на языке Objective-C

```
// Простая программа для работы с дробями, - версия Objective-C.
#import <stdio.h>
#import <objc/Object.h>
//----- @interface section -----
@interface Fraction: Object
{
    int numerator;
    int denominator;
}
-(void) setNumerator: (int) n;
-(void) setDenominator: (int) d;
-(void) print;
@end
//----- @implementation section -----
@implementation Fraction;
// getters
-(int) numerator
{
    return numerator;
}
-(int) denominator
{
    return denominator;
}
// setters
-(void) setNumerator: (int) num
{
    numerator = num;
}
-(void) setDenominator: (int) denom
{
    denominator = denom;
}
// other
-(void) print
{
    printf ("The value of the fraction is %i/%i\n", numerator,
denominator);
}
@end
//----- Листинг section -----
int main (void)
{
    Fraction *myFract;
    myFract = [Fraction new];
    [myFract setNumerator: 1];
    [myFract setDenominator: 3];
    printf ("The numerator is %i, and the denominator is %i\n",
        [myFract numerator], [myFract denominator]);
    [myFract print];      // Используем метод для отображения дроби.
    [myFract free];
    return 0;
}
```

Листинг 19.2. Вывод

The numerator is 1, and the denominator is 3
The value of the fraction is 1/3

Как можно видеть из комментариев, программа логически разбита на три раздела: раздел интерфейсов (@interface section), раздел реализации (@implementation section) и раздел программы. Обычно разделы помещаются в отдельные файлы. Раздел интерфейсов обычно помещается в заголовочный файл, который может быть включен в любую программу, которая должна работать с этим классом. Таким образом компилятору сообщается, что эти переменные и методы являются членами класса.

Раздел реализации содержит реальный код, который реализует логику работы этих методов.

Наконец, раздел программы содержит программный код, который реализует логику работы всей программы.

Имя нового класса задано как Fraction, а его родительский класс называется Object. Классы наследуют методы и переменные своих родительских классов.

Как можно видеть из раздела интерфейсов, объявления

```
int numerator;  
int denominator;
```

свидетельствуют о том, что объект Fraction содержит два целочисленных члена с именами numerator и denominator.

Члены, объявленные в этом разделе, являются переменными экземпляра. Каждый раз, когда вы создаете новый объект, создается новый и уникальный набор переменных экземпляра. Следовательно, если вы имеете две дроби, одну с именем fracA, и вторую — с именем fracB, то каждая из них имеет свой собственный числитель (numerator) и знаменатель (denominator).

Также необходимо определить методы для работы с дробями. Необходимо иметь возможность задавать значения дроби для каждой отдельной переменной. Поскольку в данном случае нет непосредственного доступа к внутреннему содержимому дроби (другими словами, нет непосредственного доступа к переменным экземпляра), т.е. необходимо написать методы, которые будут устанавливать значения для числителя и знаменателя и которые в данном случае называют *установочными* (setters). Также необходимы методы для извлечения значений из переменных экземпляра, которые называют *извлекающими* (getters).

В действительности скрытие переменных экземпляра от пользователя является еще одной ключевой концепцией ООП, которая называется *инкапсуляцией данных*. Это гарантирует неизменность поведения класса при расширении или модификации класса, т.к. код, который производит доступ к переменным класса, находится в его методах. Инкапсуляция данных способствует созданию промежуточного уровня между программистом и разработчиком класса.

Ниже приводится пример установочного метода, который может быть объявлен следующим образом.

```
-(int) numerator;
```

Предшествующий знак минус (-) говорит о том, что это метод экземпляра. Может использоваться и знак плюс (+), который будет говорить о том, что это метод класса. Метод класса — это такой метод, который выполняет операции только в самом классе, например, такие как создание экземпляра класса. Это подобно заводу, производящему

автомобили, который в нашем случае можно рассматривать как класс. Именно методы, создающие новый автомобиль, являются методами класса.

Методы экземпляра выполняют операции только для отдельного экземпляра класса, такие как установка его значений, извлечение значений переменных, отображение значений и т.д. Опять обращаясь к аналогии с автомобилями, можно сказать, что после того, как автомобиль был создан с помощью методов класса, его необходимо заправить бензином. Операция заправки бензином выполняется для конкретного автомобиля отдельно, поэтому это будет метод экземпляра.

Когда вы объявляете новый метод (что подобно объявлению функции), вы сообщаете компилятору языка Objective-C, будет ли метод возвращать значение. И если да, то значение какого типа он будет возвращать? Это можно сделать, поместив возвращаемый тип в круглые скобки после знака минус или плюс. Поэтому объявление

```
-(int) numerator;
```

свидетельствует о том, что метод экземпляра с именем `numerator` возвращает целочисленное значение. Аналогично, в строке

```
-(void) setNumerator: (int) num;
```

объявляется метод, который не возвращает никакого значения и который можно использовать при задании значения для числителя дроби.

Если метод должен принимать аргументы, он дополняется двоеточием, которое ставится за именем метода при вызове последнего. Поэтому методы для задания значений должны записываться с двоеточием: `setNumerator:` и `setDenominator:`, что говорит о том, что эти методы принимают аргументы. Если не поставить двоеточия после имени метода, то это будет говорить о том, что данные методы не принимают никаких аргументов.

Метод `setNumerator:` принимает целочисленное значение, которое обозначено именем `num`, и сохраняет его в переменной экземпляра с именем `numerator`. Аналогично, метод `setDenominator:` сохраняет значение аргумента `denom` в переменной экземпляра с именем `denominator`. Обратите внимание, что эти методы имеют непосредственный доступ к соответствующим переменным экземпляра.

Последний метод, объявленный в программе на языке Objective-C, называется `print`. Он используется для отображения значения дробного числа. Как можно видеть, этот метод не принимает аргументов и ничего не возвращает. Метод `print` используется для отображения значений числителя и знаменателя, разделенных обратной чертой. Внутри метода `main` объявлена переменная с именем `myFract` следующим образом.

```
Fraction *myFract;
```

Такое объявление свидетельствует о том, что переменная `myFract` является объектом типа `Fraction`. Таким образом, переменная `myFract` используется для хранения значений в соответствии с типами класса `Fraction`. В действительности, она указывает на структуру, которая содержит данные для отдельного экземпляра класса `Fraction`.

Теперь у вас есть возможность иметь объект типа `Fraction`, но сначала его необходимо создать. Это подобно тому, как если бы отправить на завод заявку на изготовление автомобиля. Создать объект можно с помощью следующего утверждения.

```
myFract = [Fraction new];
```

В этом выражении вы просите компилятор выделить память для хранения значения дробного числа. Выражение


```
[Fraction new]
```

посылает сообщение во вновь созданный класс `Fraction`. Вы запрашиваете класс `Fraction` на предмет того, чтобы он использовал метод для создания экземпляра класса, но вы сами не описывали этот метод. Так откуда класс возьмет этот метод? Этот метод будет взят из родительского класса.

Сейчас уже можно задать значения для объекта типа `Fraction`. Это выполнят строки программы.

```
[myFract setNumerator: 1];  
[myFract setDenominator: 3];
```

В первом утверждении посылается установочное сообщение `setNumerator:` в объект `myFract`. Аргументом для этого сообщения является целочисленное значение 1. При этом будет использован метод `setNumerator:`, который вы объявили для класса `Fraction`. Исполняющей системе для языка Objective-C известно, методы какого класса необходимо использовать, поскольку она “знает”, что объект `myFract` произведен из класса `Fraction`.

В самом методе `setNumerator:` всего одна строка, в которой передаваемое через аргумент значение присваивается переменной экземпляра с именем `numerator`. Таким образом вы можете эффективно установить для числителя объекта `myFract` значение, равное 1.

Сообщение, которое вызывает метод `setDenominator:` для объекта `myFract`, написанное в следующей строке, работает аналогичным образом.

Когда значение дроби будет установлено, в программе из листинга 19.2 вызываются два извлекающих метода `numerator` и `denominator` для извлечения значения переменных из соответствующего объекта типа `myFract`. Результат передается в метод `printf` для отображения на экране.

Затем вызывается метод `printf`. Он отображает значение дроби, которое было передано в этот метод. Просматривая программу, вы поймете, как значения числителя и знаменателя могут быть получены с помощью извлекающих методов. Дополнительный метод `print` добавлен в класс `Fraction` в целях наглядности.

Последнее утверждение программы

```
[myFract free];
```

освобождает память, которая была занята объектом типа `Fraction`.

Определение класса на языке C++ для работы с дробями

В программе из листинга 19.3 показано, как можно программно реализовать класс `Fraction` с помощью языка программирования C++. Язык C++ очень популярен среди разработчиков программного обеспечения. Он был придуман Бьярном Страуструпом из фирмы Bell Laboratories и стал первым объектно-ориентированным языком программирования на основе языка C.

Листинг 19.3. Работа с дробями на языке C++

```
#include <iostream>  
class Fraction  
{  
private:  
    int numerator;
```

```

    int denominator;
public:
    void setNumerator (int num);
    void setDenominator (int denom);
    int Numerator (void);
    int Denominator (void);
    void print (Fraction f);
};
void Fraction::setNumerator (int num)
{
    numerator = num;
}
void Fraction::setDenominator (int denom)
{
    denominator = denom;
}
int Fraction::Numerator (void)
{
    return numerator;
}
int Fraction::Denominator (void)
{
    return denominator;
}
void Fraction::print (void)
{
    std::cout << "The value of the fraction is " << numerator << '/'
        << denominator << '\n';
}

int main (void)
{
    Fraction myFract;
    myFract.setNumerator (1);
    myFract.setDenominator (3);
    myFract.print ();
    return 0;
}

```

Листинг 19.3. Вывод

The value of the fraction is 1/3

В соответствии с синтаксисом языка C++, члены (переменные экземпляра) `numerator` и `denominator` помечены как `private` для того, чтобы реализовать инкапсуляцию данных, что означает защиту данных от прямого доступа из-за пределов класса. Метод `setNumerator` объявлен следующим образом.

```
void Fraction::setNumerator (int num)
```

Этому методу предшествует обозначение `Fraction::` для того, чтобы показать отношение этого метода к классу `Fraction`. Новый экземпляр класса `Fraction` создается так, как объявляются переменные в языке C, что показано в следующем объявлении, сделанном в классе `main`.

```
Fraction myFract;
```

Значения для числителя и знаменателя дроби устанавливаются как числа 1 и 3 соответственно, с помощью вызовов следующих методов.

```
myFract.setNumerator (1);  
myFract.setDenominator (3);
```

Затем значение дроби отображается с помощью метода `print`. Достаточно сложное выражение используется внутри метода `print`, которое выглядит следующим образом.

```
std::cout << "The value of the fraction is " << numerator << '/'  
<< denominator << '\n';
```

Здесь аббревиатура `cout` является именем стандартного выходного потока, аналогичного потоку `stdout` в языке C. Оператор "`<<`" является *оператором вывода в поток* и обеспечивает легкий способ получения вывода. Вспомните, что оператор "`<<`" в языке C является оператором сдвига. Это является довольно важным аспектом языка C++ — возможность использовать *перегрузку операторов*, что позволяет задать действия операторов для отдельных классов. В данном случае именно оператор сдвига влево перегружен так, что в данном контексте (если в качестве левого операнда используется поток) он вызывает метод для записи форматированных значений в выходной поток вместо того, чтобы выполнять операцию сдвига влево.

Приведу еще один пример перегрузки. Возможно вы захотите перегрузить оператор сложения "`+`" для того, чтобы он мог суммировать дробные значения, как показано ниже.

```
myFract + myFract2
```

При этом будет вызываться соответствующий метод класса `Fraction`, с тем чтобы корректно выполнить сложение. Каждое выражение, которое следует за оператором "`<<`", обрабатывается и записывается в стандартный выходной поток. В нашем случае сначала будет выведена строка "The value of the fraction is", за ней — значение числителя с последующей косой чертой и значение знаменателя. После этого будет сделан переход на новую строку.

Язык C++ обладает большим набором различных возможностей. В связи с этим ознакомьтесь с рекомендациями относительно выбора хорошего учебника в приложении А, "Ресурсы". Обратите внимание, что в предыдущем примере методы для получения значений `Numerator()` и `Denominator()` были определены в классе `Fraction`, но не были использованы.

Определение класса на языке C# для работы с дробями

Рассмотрим последний пример в этой главе. Программа из листинга 19.4 показывает использование дробей с помощью языка программирования C#. Этот язык разработан фирмой Microsoft и является самым современным и очень популярным объектно-ориентированным языком, поскольку он в наибольшей степени приспособлен для разработки приложений с помощью среды .NET.

Листинг 19.4. Работа с дробями на языке C#

```
using System;
class Fraction
{
    private int numerator;
    private int denominator;
    public int Numerator
    {
        get
        {
            return numerator;
        }
        set
        {
            numerator = value;
        }
    }
    public int Denominator
    {
        get
        {
            return denominator;
        }
        set
        {
            denominator = value;
        }
    }
    public void print ()
    {
        Console.WriteLine("The value of the fraction is {0}/{1}",
            numerator, denominator);
    }
}

class example
{
    public static void Main()
    {
        Fraction myFract = new Fraction();
        myFract.Numerator = 1;
        myFract.Denominator = 3;
        myFract.print ();
    }
}
```

Листинг 19.4. Вывод

The value of the fraction is 1/3

Как видите, язык программирования C# лишь незначительно отличается от других объектно-ориентированных языков. Описание класса Fraction начинается с объявления двух переменных numerator и denominator как закрытых членов, о чем свидетельствует ключевое слово private.

Члены `Numerator` и `Denominator` имеют методы доступа к переменным и называются *свойствами*. Рассмотрим более подробно свойство `Numerator`.

```
public int Numerator
{
    get
    {
        return numerator;
    }
    set
    {
        numerator = value;
    }
}
```

Код для метода `get` выполняется тогда, когда в выражении требуется получить значение числителя.

```
num = myFract.Numerator;
```

Код для метода `set` выполняется тогда, когда в выражении требуется присвоить значение числителю.

```
myFract.Numerator = 1;
```

Действительное значение, которое присваивается при вызове метода, сохраняется в переменной с именем `value`. Круглые скобки с этими методами не используются.

Вполне понятно, что можно объявить методы, которые будут принимать аргументы. Например, на языке `C#` можно определить метод, который можно использовать для присваивания значения $2/5$ дроби с помощью только одного вызова.

```
myFract.setNumAndDen (2, 5)
```

Возвращаясь к программе из листинга 19.4 рассмотрим утверждение

```
Fraction myFract = new Fraction();
```

которое используется для создания нового экземпляра класса `Fraction` и присваивания созданного объекта переменной с именем `myFract`. Значение для объекта типа `Fraction` затем устанавливается равным $1/3$.

Метод `print` вызывается для того, чтобы отобразить это значение. Внутри метода `print` используется метод `WriteLine` из класса `Console`, который и производит вывод значений в выходной поток. Аналогично символу `"%"`, используемому ранее в методе `printf`, символы `{0}` используются для того, чтобы отобразить первое значение, символы `{1}` используются для отображения второго значения и т.д.

Как и в примере с использованием языка `C++`, здесь не используются методы доступа из класса `Fraction`. Они также включены в целях наглядности.

На этом заканчивается краткое введение в объектно-ориентированное программирование. Надеюсь, прочитав эту главу, вы сможете лучше уяснить особенности объектно-ориентированного программирования в сравнении с тем, которое используется в языке `C`.

В главе было показано, как можно написать простую программу на трех объектно-ориентированных языках для работы с дробными числами. Если вас заинтересовали классы и вы собираетесь расширить возможности работы с дробями, то вы можете добавить, например, такие операции, как сложение, вычитание, умножение, деление, инверсия и приведение дробей.

А

Справочник по языку С

В данной главе описываются все возможности языка программирования С в формате, удобном для получения быстрой справки. Рекомендуем последовательно прочитать разделы этого приложения, что позволит вам не только закрепить материал, изученный в главах книги, но и будет способствовать более глубокому общему восприятию языка С. Материал данного приложения основывается на стандарте ANSI C99 (ISO/IEC 9899:1999).

1.0. Диграфы и идентификаторы

1.1. Символьные диграфы

В табл. А.1 перечислены специальные двухсимвольные последовательности (диграфы), которые эквивалентны соответствующим односимвольным знакам пунктуации.

Таблица А.1. Символьные диграфы

Диграф	Значение
<:	{
:>	}
<%	{
%>	}
%:	#
%:%:	##

1.2 Идентификаторы

В языке С идентификатор состоит из последовательности букв (в нижнем или верхнем регистре), универсальных символьных имен (раздел 1.2.1), цифр и символов подчеркивания. Первый символ в идентификаторе обязательно должен быть буквой, символом подчеркивания или универсальным символьным именем. От начала

идентификатора 31 символ являются значимыми для внешнего имени, а первые 63 символа являются значимыми для внутренних идентификаторов и имен макроопределений.

1.2.1. Универсальные символьные имена

Универсальное символьное имя формируется с помощью символов `\u`, за которыми следуют четыре шестнадцатеричных числа, или с помощью символов `\U`, за которыми следуют восемь шестнадцатеричных чисел. Если первый символ идентификатора задан универсальным символом, его значение не может быть цифровым символом. При использовании имен в идентификаторах, универсальные символы также не должны определять символы, значение которых меньше чем $A0_{16}$ (в отдельных случаях меньше чем 24_{16} , 40_{16} или 60_{16}), или символы в диапазоне от $D800_{16}$ до $DFFF_{16}$ включительно.

Универсальные символьные имена можно использовать в качестве имен, символьных констант или символьных строк.

1.2.2. Ключевые слова

Идентификаторы, перечисленные в табл. А.2, являются ключевыми словами, которые имеют специальное значение для компилятора C.

Таблица А.2. Ключевые слова

<code>_Bool</code>	<code>default</code>	<code>if</code>	<code>sizeof</code>	<code>while</code>
<code>_Complex</code>	<code>do</code>	<code>inline</code>	<code>static</code>	
<code>_Imaginary</code>	<code>double</code>	<code>int</code>	<code>struct</code>	
<code>auto</code>	<code>else</code>	<code>long</code>	<code>switch</code>	
<code>break</code>	<code>enum</code>	<code>register</code>	<code>typedef</code>	
<code>case</code>	<code>extern</code>	<code>restrict</code>	<code>union</code>	
<code>char</code>	<code>float</code>	<code>return</code>	<code>unsigned</code>	
<code>const</code>	<code>for</code>	<code>short</code>	<code>void</code>	
<code>continue</code>	<code>goto</code>	<code>signed</code>	<code>volatile</code>	

2.0. Комментарии

Комментарий можно вставить в программу двумя способами. Например, он может начинаться двумя косыми чертами (`//`). При этом все символы, которые следуют за этими чертами до конца строки, игнорируются компилятором. Комментарий также можно вставить с помощью символов `/*`, задающих начало комментария, и символов `*/`, которыми комментарий заканчивается. Любые символы, которые заключены в такие символы комментариев, игнорируются компилятором, даже если они располагаются на нескольких строках. Комментарии могут использоваться в любом месте программы, где разрешены символы пробела. Но комментарии не могут быть вложенными, т.е. первая встретившаяся последовательность символов `*/` означает конец комментария, независимо от того, сколько последовательностей символов `/*` было до этого.

3.0. Константы

3.1. Целочисленные константы

Целочисленная константа представляет собой последовательность цифр с предшествующим знаком плюс или минус. Если первой цифрой является 0, то целое число считается восьмеричным и в этом случае все последующие цифры должны быть от 0 до 7. Если первой цифрой является 0 и за ней непосредственно следует буква x (или X), то целое число считается шестнадцатеричной константой и все последующие символы должны быть от 0 до 9 и от a до f (или от A до F).

К десятичной целочисленной константе в конце можно добавлять буквы l или L, что говорит о том, что данная константа имеет тип `long int`. Если заданное значение не может поместиться в тип `long int`, то оно будет трактоваться как тип `long long int`. Если буквы l или L добавлены в конце восьмеричной или шестнадцатеричной константы, то они также принимают тип `long int`. Если заданное значение не может поместиться в тип `long int`, то оно будет трактоваться как тип `long long int`.

Наконец, если заданное значение не может поместиться в тип `long long int`, то оно будет трактоваться как тип `unsigned long long int`.

Суффиксы ll или LL также можно добавлять к десятичным целочисленным константам, с тем, чтобы преобразовать их в тип `long int`. Когда эти суффиксы добавлены в конце восьмеричной или шестнадцатеричной константы, то они также принимают тип `long long int`. Если заданное значение не может поместиться в тип `long long int`, то оно будет трактоваться как тип `unsigned long long int`.

Суффиксы u или U также можно добавлять к целочисленным константам для того, чтобы преобразовать их в беззнаковый тип `unsigned`. Если заданное значение не может поместиться в тип `unsigned int`, то оно преобразовывается в тип `unsigned long int`. Если заданное значение не может поместиться и в тип `unsigned long int`, то оно преобразовывается в тип `unsigned long long int`.

Оба суффикса, `unsigned` и `long`, могут быть добавлены к целочисленной константе, с тем, чтобы получить тип `unsigned long int`. Если значение константы велико для типа `unsigned long int`, то он будет преобразован в тип `unsigned long long int`.

Оба суффикса, `unsigned` и `long long`, могут быть добавлены к целочисленной константе, с тем, чтобы получить тип `unsigned long long int`.

Если десятичная целочисленная константа без суффикса имеет значение, превышающее максимальное значение для типа `signed int`, она будет трактоваться как `long int`. Если значение превышает максимальное значение для типа `long int`, она будет трактоваться как тип `long long int`.

Если восьмеричная или шестнадцатеричная константа без суффикса имеет значение, большее, чем это допустимо для типа `signed int`, она будет трактоваться как `unsigned int`. Если значение больше, чем это допустимо для типа `unsigned int`, он будет трактоваться как `long int`, а если значение больше, чем значение, допустимое для типа `long int`, она будет трактоваться как `unsigned long int`. Если значение больше, чем значение, допустимое для типа `unsigned long int`, она будет трактоваться как `long long int`. Наконец, если значение больше, чем это допустимо для типа `long long int`, она будет трактоваться как `unsigned long long int`.

3.2. Вещественные константы

Вещественные константы состоят из последовательности десятичных цифр, десятичной точки и еще одной последовательности десятичных цифр. Знак минус может предшествовать значению для получения отрицательных величин. Любая последовательность цифр до десятичной точки или после нее может быть пропущена, но не обе вместе.

Если непосредственно за вещественной константой находится буква *e* (или *E*) и дополнительно число со знаком, то константа выражена в научной нотации. Это дополнительное число со знаком (экспонента) представляет степень 10, на которую умножается число, предшествующее букве *e* (мантисса). Например, $1.5e-2$ представляет значение $1.5 \cdot 10^{-2}$, или $.015$.

Шестнадцатеричная вещественная константа состоит из префикса *0x* или *0X*, за которыми непосредственно следует одна или несколько шестнадцатеричных цифр, за которыми дополнительно может находиться буква *p* или *P* с дополнительной двоичной экспонентой. Например, $0x3p10$ представляет значение $3 \cdot 2^{10}$.

Вещественные константы трактуются как значения типа *double*. Дополнительная буква суффикса *f* или *F* может быть добавлена для получения константы типа *float* вместо типа *double*.

Дополнительная буква суффикса *l* или *L* может быть добавлена для получения константы типа *long double*.

3.3. Символьные константы

Символы, помещенные в одиночные кавычки, считаются символьной константой. В зависимости от реализации, можно ограничивать одиночными кавычками один символ. Универсальные символы (раздел 1.2.1) могут использоваться в символьных константах для использования символа, не включенного в стандартный набор символов.

3.3.1. Управляющие последовательности

Существуют специальные управляющие последовательности, которые начинаются с символа обратной черты. Такие управляющие последовательности представлены в табл. А.3.

Таблица А.3. Специальные управляющие последовательности

Символ	Значение
<code>\a</code>	Предупреждающий звуковой сигнал
<code>\b</code>	Возврат
<code>\f</code>	Подача страницы
<code>\n</code>	Новая строка
<code>\r</code>	Возврат каретки
<code>\t</code>	Горизонтальная табуляция
<code>\v</code>	Вертикальная табуляция
<code>\\</code>	Обратная черта
<code>\"</code>	Двойные кавычки
<code>\'</code>	Одинарные кавычки

Символ	Значение
\?	Знак вопроса
\nnn	Восьмеричное значение символа
\unnnn	Универсальное символьное имя
\Unnnnnnnnn	Универсальное символьное имя
\xnn	Шестнадцатеричное значение символа

При восьмеричном значении символа, может использоваться один из трех восьмеричных символов. В остальных трех случаях используются шестнадцатеричные символы.

3.3.2. Символьные константы Unicode

Символьные константы Unicode записываются как `L'x'`. Типом такой константы будет `wchar_t`, который определен в стандартном заголовочном файле `stddef.h`. Символьные константы Unicode представляют возможность перейти к набору символов, который не может быть представлен обычным типом `char`.

3.4. Строки символов

Последовательность из нуля или более символов, ограниченная двойными кавычками, представляет собой строку символов. Любой разрешенный символ может находиться в строке символов, включая управляющие последовательности символов, перечисленные выше. Компилятор автоматически вставляет нулевой символ (`'\0'`) в конец строки символов.

Обычно компилятор создает указатель на первый символ строки типа "указатель на `char`". Но, когда символьная константа представлена с оператором `sizeof` для инициализации массива символов или с оператором `&`, тип строковой константы трактуется как "массив `char`".

Строки символов в программе нельзя модифицировать.

3.4.1. Объединение строк символов

Препроцессор автоматически объединяет смежные строковые константы. Строки могут находиться рядом или могут быть разделены одним или несколькими пробелами. Поэтому следующие три отдельные строки

```
"a" " character "
"string"
```

будут преобразованы в следующую строку

```
"a character string"
```

3.4.2. Многобайтные символы

В зависимости от реализации, последовательности символов могут сдвигаться вперед или назад в строке символов, и при этом используется понятие многобайтового символа.

3.4.3 Строковые константы Unicode

Строки символов из расширенного набора символов записываются с помощью формата `L" . . . "`. Типом такой строковой константы будет "указатель на `wchar_t`", где тип `wchar_t` определен в заголовочном файле `stddef.h`.

3.5. Перечислимые константы

Идентификаторы, которые используются при объявлении перечислимого типа, являются константами этого типа, но могут трактоваться компилятором как тип `int`.

4.0. Типы данных и их объявления

В этом разделе кратко описываются основные типы данных, унаследованные типы данных, перечислимые типы и директива `typedef`. Также приводится описание форматов при объявлении переменных.

4.1 Объявления

Когда описывается отдельная структура, объединение, перечислимый тип или используется директива `typedef`, компилятор не выделяет автоматически память. Описание просто служит напоминанием компилятору об отдельном типе данных, который связывается (не обязательно) с некоторым именем. Такое описание можно выполнять как в функции, так и за ее пределами. В первом случае только функция "знает" об этих переменных, во втором случае об этой переменной будет известно во всем оставшемся участке файла.

После описания типа можно объявлять переменную этого типа данных. При объявлении переменной любого типа данных для нее резервируется память, за исключением объявления переменных с ключевым словом `extern`, когда память может не выделяться (см. раздел 6.0).

Компилятор может распределять память для переменных одновременно с описанием отдельной структуры, объединения или перечислимого типа. Для этого необходимо просто перечислить переменные, прежде чем будет поставлена заключительная точка с запятой.

4.2. Стандартные типы данных

Основные типы данных описаны в табл. А.4. Переменные могут быть объявлены как любой основной тип в соответствии со следующим форматом.

```
тип имя = инициализирующее_значение;
```

Присваивание инициализирующего значения переменной является дополнительным и выполняется в соответствии с правилами, описанными в разделе 6.2. Можно объявлять более одной переменной с использованием следующего общего формата.

```
тип имя = инициализирующее_значение, имя = инициализирующее_значение, . . . ;
```

Перед объявление типа можно дополнительно указать класс памяти, что более подробно рассмотрено в разделе 6.2. Если задан класс памяти и тип переменной объявлен как `int`, то ключевое слово `int` можно пропустить. Например, в объявлении

```
static counter;
```

задается переменная `counter` типа `static int`.

Таблица А.4. Стандартные типы данных

Тип	Описание
<code>int</code>	Целочисленное значение, т.е. оно не содержит десятичной точки и гарантированно содержит по крайней мере 16 битов точности
<code>short int</code>	Целочисленное значение пониженной точности. На некоторых машинах занимает вдвое меньше памяти, чем тип <code>int</code> . Гарантированно содержит по крайней мере 16 битов точности
<code>long int</code>	Целочисленное значение повышенной точности. Гарантированно содержит по крайней мере 32 бита точности
<code>long long int</code>	Целочисленное значение экстраповышенной точности. Гарантированно содержит по крайней мере 64 бита точности
<code>unsigned int</code>	Целочисленное положительное значение. Может хранить положительное значение, вдвое большее, чем тип <code>int</code> . Гарантированно содержит по крайней мере 16 битов точности
<code>Float</code>	Число с плавающей запятой, т.е. число, включающее десятичные разряды. Гарантированно содержит по крайней мере шесть цифр точности
<code>Double</code>	Число с плавающей запятой повышенной точности. Гарантированно содержит по крайней мере 10 цифр точности
<code>long double</code>	Число с плавающей запятой экстраповышенной точности. Гарантированно содержит по крайней мере 10 цифр точности
<code>Char</code>	Одиночный символ. В некоторых системах может дополняться знаком, если используется в выражениях
<code>unsigned char</code>	То же самое, что и <code>Char</code> , за исключением того, что не может использоваться со знаком
<code>signed char</code>	То же самое, что и <code>Char</code> , за исключением того, что может использоваться со знаком
<code>_Bool</code>	Булев тип. Размер должен быть достаточен для хранения значений 0 и 1
<code>float _Complex</code>	Комплексное число
<code>double _Complex</code>	Комплексное число повышенной точности
<code>long double _Complex</code>	Комплексное число экстраповышенной точности
<code>Void</code>	Отсутствие типа. Используется для обозначения того, что функция не должна возвращать значение, или для явной отмены результата расчета выражения. Также используется как обобщающий указатель (<code>void *</code>).

Обратите внимание, что модификатор `signed` может также помещаться и перед типами `short int`, `int`, `long int` и `long long int`. Но поскольку эти типы имеют знак по умолчанию, это не производит никакого эффекта.

Типы данных `_Complex` и `_Imaginary` позволяют объявлять комплексные и мнимые переменные и производить над ними арифметические вычисления, для чего используются функции из стандартных библиотек. Обычно для работы с этими типами к программам необходимо подключить заголовочный файл `complex.h`, в котором описываются макроопределения и функции для работы с комплексными и мнимыми числами.

Например, переменная `c1` типа `double _Complex` может быть объявлена и инициализирована значением `5+10.5i` с помощью следующего утверждения:

```
double _Complex c1 = 5 + 10.5 * I;
```

Библиотечные подпрограммы, такие как `creal` и `cimag`, могут использоваться для выделения целой и мнимой частей соответственно.

В стандартной реализации не требуется поддержка типов `_Complex` и `_Imaginary`, хотя может поддерживаться один из этих типов или оба типа.

Заголовочный файл `stdbool.h` может подключаться к программе для более удобной работы с булевыми типами. В этом файле описаны макроопределения `bool`, `true` и `false`, позволяющие записывать утверждения следующим образом.

```
bool endOfData = false;
```

4.3. Производные типы данных

Производными типами данных являются такие типы, которые построены на основе одного или нескольких базовых типов. Производными типами являются массивы, структуры, объединения и указатели. Функции, которые возвращают значения, также рассматриваются как производный тип данных. Все они, за исключением функций, описаны в следующих разделах. Функции рассматриваются отдельно в разделе 7.0.

4.3.1. Массивы

Одномерные массивы

Массив можно описать таким образом, чтобы он содержал основные типы или любые производные типы данных. Массивы функций не разрешены (хотя можно использовать массивы указателей на функции).

Объявление массива производится в соответствии со следующим форматом.

```
тип имя[n] = { инициализирующее_выражение, инициализирующее_
выражение, ... };
```

Выражение `n` задает количество элементов массива `имя` и может быть опущено, если задан список инициализирующих элементов. В этом случае размер массива определяется по количеству инициализирующих значений или по номеру наибольшего индекса элемента, на который можно сослаться, если с инициализирующими значениями используется индексы.

При описании глобальных массивов инициализирующее значение может быть константным выражением. Инициализирующих значений может быть меньше, чем элементов массива, но не больше. Если инициализирующих значений меньше, чем элементов массива, то будут инициализированы только те первые элементы, для которых заданы инициализирующие значения, а оставшиеся элементы будут установлены в 0.

При инициализации массива символов можно использовать специальный вид инициализации и инициализировать массив строковой константой. Например, в следующем утверждении

```
char today[] = "Monday";
```

объявлен массив символов `today`, который инициализирован символами 'M', 'o', 'n', 'd', 'a', 'y' и '\0' соответственно.

Если в массиве символов явно не оставлено места для заключающего нулевого символа, то компилятор не будет помещать нулевой символ в конец строки.

```
char today[6] = "Monday";
```

Такое объявление массива символов `today`, в котором выделено место только для шести символов, заполнит массив следующим образом: 'M', 'o', 'n', 'd', 'a' и 'y' соответственно.

Помещая номера элементов в квадратных скобках, можно инициализировать элементы массива в любом порядке. Например, при объявлении

```
int x = 1233;  
int a[] = { [9] = x + 1, [3] = 3, [2] = 2, [1] = 1 };
```

будет создан массив из 10 элементов с именем `a` (размер задается по максимальному индексу) и последний элемент будет инициализирован значением `x+1` (1234), а первые три значениями 1, 2 и 3 соответственно.

4.3.1.1. Массивы переменного размера

Внутри функции или блока можно задавать размер массива с помощью выражений, содержащих переменные. В этом случае размер массива вычисляется во время выполнения программы. Например, в функции

```
int makeVals (int n)  
{  
    int valArray[n];  
    ...  
}
```

объявляется автоматически массив с именем `valArray` и размером `n` элементов, где значение `n` вычисляется во время выполнения и может изменяться при отдельных вызовах этой функции. Массивы переменного размера нельзя инициализировать во время объявления.

4.3.1.2 Многомерные массивы

Общий формат для объявления многомерных массивов следующий.

```
тип имя[d1][d2]...[dn] = список_инициализаторов;
```

Здесь объявляется массив `имя`, который содержит `d1*d2*...*dn` элементов указанного типа. Например, в утверждении

```
int three_d [5][2][20];
```

объявляется трехмерный массив с именем `three_d`, содержащий 200 целочисленных значений.

На отдельный элемент многомерного массива можно сослаться, указывая отдельно индекс каждого измерения в квадратных скобках. Например, утверждение

```
three_d [4][0][15] = 100;
```

поместит значение 100 в указанный элемент массива `three_d`.

Многомерные массивы можно инициализировать так же, как и одномерные массивы. Можно использовать фигурные скобки для выделения групп инициализирующих значений, принадлежащих отдельным измерениям.

Ниже объявлен двумерный массив с именем `matrix`, содержащий четыре строки и три столбца.

```
int matrix[4][3] =
{ { 1, 2, 3 },
{ 4, 5, 6 },
{ 7, 8, 9 } };
```

Элементы первой строки массива `matrix` будут инициализированы значениями 1, 2 и 3 соответственно, элементы второй строки будут инициализированы значениями 4, 5 и 6, а элементы третьей строки будут инициализированы значениями 7, 8 и 9 соответственно. Все элементы четвертой строки будут установлены в 0, поскольку для четвертой строки нет инициализирующих значений.

Объявление

```
static int matrix[4][3] =
{ 1, 2, 3, 4, 5, 6, 7, 8, 9 };
```

приведет к инициализации теми же самыми значениями массива `matrix`, который инициализируется в порядке распределения размерностей слева направо.

Объявление

```
int matrix[4][3] =
{ { 1 },
{ 4 },
{ 7 } };
```

установит для первого элемента первой строки массива `matrix` значение 1, для первого элемента второй строки значение 4 и для первого элемента третьей строки значение 7. Во все остальные элементы будет установлено значение 0 по умолчанию.

Наконец, объявление

```
int matrix[4][3] = { [0][0] = 1, [1][1] = 5, [2][2] = 9 };
```

инициализирует указанные элементы матрицы заданными значениями.

4.3.2. Структуры

Общий формат для объявления структур следующий.

```
struct name
{
    memberDeclaration
    memberDeclaration
    ...
} variableList;
```

Здесь описывается структура с именем `name`, в которой содержатся члены `member-Declaration`, также объявляемые по стандартным правилам. Каждое объявление состоит из указания типа, за которым следует список из одного или нескольких имен.

Переменные могут объявляться во время описания структуры простым перечислением имен перед заключительной точкой с запятой, или их можно объявлять последовательно, используя следующий формат.

```
struct name variableList;
```

Такой формат не может использоваться в том случае, когда опускается имя структуры `name`. В этом случае все необходимые переменные должны быть объявлены во время описания структуры.

Способ инициализации структуры аналогичен тому, который используется при инициализации массивов. Члены структуры можно инициализировать, помещая инициализирующие значения в круглые скобки. Каждое значение может быть выражено константным выражением, если описывается глобальная структура.

Объявление

```
struct point
{
    float x;
    float y;
} start = {100.0, 200.0};
```

задает структуру с именем `point` и переменную `start` типа `struct point`, которая инициализируется заданными значениями. Отдельные члены можно инициализировать в любом порядке с помощью специальной записи

```
.member = value
```

в списке инициализаторов.

```
struct point end = { .y = 500, .x = 200 };
```

Объявление

```
struct entry
{
    char *word;
    char *def;
} dictionary[1000] = {
    { "a", "first letter of the alphabet" },
    { "aardvark", "a burrowing African mammal" },
    { "aback", "to startle" }
};
```

приводит к созданию массива с именем `dictionary`, содержащим 1000 элементов типа `entry`, первые три из которых инициализированы заданными строками символов. Используя инициализацию с помощью индексов, можно то же самое задать следующим образом.

```
struct entry
{
    char *word;
    char *def;
```



```

} dictionary[1000] = {
    [0].word = "a", [0].def = "first letter of the alphabet",
    [1].word = "aardvark", [1].def = "a burrowing African mammal",
    [2].word = "aback", [2].def = "to startle"
};

```

Или инициализировать так, что это будет аналогично вышеприведенной инициализации.

```

struct entry
{
    char *word;
    char *def;
} dictionary[1000] = {
    { { .word = "a", .def = "first letter of the alphabet" },
      { .word = "aardvark", .def = "a burrowing African mammal" } ,
      { .word = "aback", .def = "to startle" }
};

```

Автоматические структурные переменные могут инициализироваться другой структурой того же типа следующим образом.

```

struct date tomorrow = today;

```

Здесь объявляется структура с именем `tomorrow` типа `date`, которой присваивается содержимое структуры `today` (предварительно объявленной) типа `date`.

Объявление членов структуры `memberDeclaration` может производиться в соответствии со следующим форматом

```

type fieldName : n

```

где объявляется поле `fieldName`, занимающее `n` битов структуры, при этом `n` является целым числом. Поля могут упаковываться слева направо (в некоторых машинах справа налево). Если имя `fieldName` пропущено, то заданное число резервируется в структуре, но на них нельзя ссылаться.

Если имя `fieldName` пропущено и `n` равно 0, то поле следующего блока выравнивается по границе блока, где размер блока зависит от реализации. Типом поля может быть `_Bool`, `int`, `signed int` или `unsigned int`. В зависимости от реализации, поле типа `int` может трактоваться как `signed` или `unsigned`. Оператор адреса (`&`) не может использоваться с полями, а также нельзя задать массивы полей.

4.3.3. Объединения

Общий формат для объявления объединений следующий.

```

union name
{
    memberDeclaration
    memberDeclaration
    ...
} variableList;

```

Здесь описывается объединение с именем `name` и членами, которые представлены словами `memberDeclaration`. Каждый член объединения использует одно и то же пространство памяти и компилятор производит выделение памяти в соответствии с самым большим членом объединения.

Переменные типа объединение можно объявлять одновременно с объявление и объединения, или они могут быть в дальнейшем объявлены в соответствии с форматом

```
union name variableList;
```

где после имени объединения указывается имя переменной. Программист должен позаботиться о том, чтобы значение, извлекаемое из объединения, не противоречило типу переменной. Первый член объединения может быть инициализирован с помощью значения, заключенного в фигурные скобки. При объявлении глобальной переменной это может быть константное выражение.

```
union shared
{
    long long int l;
    long int w[2];
} swap = { 0xffffffff };
```

Выше объявлена переменная с именем swap типа объединение, инициализирующее значение для которой равно ffffffff. Вместо первого члена можно задать значение с помощью других членов.

```
union shared swap2 = { .w[0] = 0x0, .w[1] = 0xffffffff; }
```

Автоматические переменные типа объединение также можно инициализировать объединением такого же типа.

```
union shared swap2 = swap;
```

4.3.4. Указатели

Базовый формат для объявления указателей следующий.

```
type *name;
```

Имя name объявлено как тип "указатель на type", и при этом тип может быть любым из основных базовых типов или производным типом. Например, в утверждении

```
int *ip;
```

объявляется указатель ip на тип int, а объявление

```
struct entry *ep;
```

задает указатель ep на структуру entry. Указатели, которые указывают на элементы массива, объявляются как указатели на элементы, составляющие массив. Например, предыдущее объявление переменной ip может также использоваться как указатель на массив целых чисел.

Разрешены и более сложные формы объявления указателей. Например, объявление

```
char *tp[100];
```

задает 100 указателей на символы, а объявление

```
struct entry (*fnPtr) (int);
```

задает указатель fnPtr на функцию, которая возвращает структуру entry и принимает один аргумент типа int.

Указатель может быть проверен на нулевое значение с помощью его сравнения с константным выражением, равным нулю.

Способ, с помощью которого производится преобразование указателя в целочисленное значение и обратно, зависит от конкретного типа компьютера, т.к. размер целочисленного значения должен вмещать указатель.

Тип "указатель на void" является обобщающим типом. В стандарте языка гарантируется, что указатель на любой тип может быть присвоен указателю на тип void. Также верно и обратное присвоение без изменения их значений.

Другие присваивания указателей на различные типы не разрешены и обычно при этом возникает предупреждающее сообщение компилятора.

4.4. Перечислимые типы данных

Общий формат для объявления перечислимых типов данных следующий.

```
enum name { enum_1, enum_2, ... } variableList;
```

Перечислимый тип с именем name объявлен с перечислимыми значениями enum_1, enum_2, ... каждое из которых является идентификатором или идентификатором с последующим знаком равенства и константным выражением.

Список переменных variableList, которые будут представлять тип enum name, является дополнительным.

Компилятор присваивает последовательно целые значения идентификаторам начиная с нуля. Если за идентификатором следует знак равенства (=) и константное выражение, то значение этого выражения связывается с идентификатором. Значения связываются с последовательностью идентификаторов, при этом для очередного идентификатора значение увеличивается на 1. Идентификаторы перечислимого типа трактуются компилятором как константные целочисленные значения.

Если необходимо объявить переменную уже объявленного (и названного) перечислимого типа, то можно использовать следующую запись.

```
enum name variableList;
```

Объявленной переменной конкретного перечислимого типа могут присваиваться значения только из этого типа, хотя компилятор может и не воспринимать ошибку невыполнение данного правила.

4.5. Утверждение typedef

Утверждение typedef используется для присваивания нового имени основному или производному типу данных. Утверждение typedef не определяет новый тип данных, а просто создает новое имя для уже существующего типа.

Поэтому переменные, объявленные с использованием нового имени, воспринимаются компилятором так, как если бы переменные были объявлены с помощью типа, используемого при создании нового имени.

Написание утверждения typedef похоже на объявление обычной переменной. За исключением того, что пишется новое имя на том месте, где ставится имя переменной. А на самом первом месте должно находиться ключевое слово typedef. Например, утверждение

```
typedef struct  
{  
    float x;
```

```
float y;  
} Point;
```

связывает с именем Point структуру, содержащую два вещественных члена с именами x и y. При объявлении новых переменных типа Point используется обычный синтаксис.

```
Point origin = { 0.0, 0.0 };
```

4.6 Модификаторы типа `const`, `volatile` и `restrict`

Ключевое слово `const` может находиться перед объявлением переменной и указывает компилятору на то, что эта переменная не может быть модифицирована. Поэтому объявление

```
const int x5 = 100;
```

говорит о том, что переменная `x5` является целочисленной константой. Это означает, что во время выполнения программы этой переменной нельзя присваивать значения. Компилятор может не реагировать на присваивание значения переменной, объявленной с ключевым словом `const`.

Модификатор `volatile` явно указывает компилятору на то, что значение переменной должно изменяться (обычно динамически). Когда такая переменная используется в выражениях, ее значение доступно в любое время.

Для объявления переменной `port17` в качестве типа “изменяемый указатель на `char`”, необходимо записать следующее

```
volatile char *port17;
```

Ключевое слово `restrict` можно использоваться с указателями, что послужит своеобразной подсказкой компилятору о возможности оптимизации (подобно ключевому слову `register` для переменных). Ключевое слово `restrict` указывает компилятору на то, что данный указатель является единственной ссылкой на определенный объект, т.е. что на этот объект больше не ссылается никакой другой указатель из этой области видимости. Строки

```
int * restrict intPtrA;  
int * restrict intPtrB;
```

говорят компилятору о том, что в области видимости, где объявлены указатели `intPtrA` и `intPtrB`, они никогда не будут указывать на один и тот же объект. В данном случае они используются для указания на целые числа (например, в массиве), которые являются взаимно исключающими.

5.0. Выражения

Имена переменных, функций, массивов, констант, вызовы функций, ссылки на массивы, структуры и объединения рассматриваются как выражения. Применяя унарный оператор (где допустимо) к одному из этих выражений, также можно получить выражение. Комбинируя два или более таких выражений с помощью бинарных или троичных операторов, тоже получим выражение. Наконец, выражение, заключенное в круглые скобки, также будет выражением.

Выражение любого типа, кроме `void`, которое идентифицирует объект данных, называется *value* (левостороннее выражение, или выражение, которое может находиться

в левой части оператора присваивания). Если ему разрешено присваивать значение, то его называют *модифицируемым левосторонним выражением* (modifiable lvalue).

Модифицируемые левосторонние выражения могут размещаться только в определенных местах. Выражение с левой стороны оператора присваивания должно быть модифицируемым левосторонним выражением. Более того, операторы инкремента и декремента могут использоваться только с модифицируемыми левосторонними выражениями, как и унарный оператор получения адреса &.

5.2. Операторы языка C

В табл. А.5 представлены различные операторы языка C. Эти операторы перечислены в порядке уменьшения приоритета. Операторы, сгруппированные вместе, имеют одинаковый приоритет.

Таблица А.5. Операторы языка C

Оператор	Описание	Ассоциативность
()	Вызов функции	Слева направо
[]	Ссылка на элемент массива	
->	Ссылка на член структуры с помощью указателя	
.	Ссылка на член структуры	
-	Унарный минус	Справа налево
+	Унарный плюс	
++	Инкремент	
--	Декремент	
!	Логическое отрицание	
~	Одинарное дополнение	
*	Ссылка на указатель	
&	Адрес	
sizeof	Размер объекта	
(type)	Приведение типов (преобразование)	
*	Умножение	Слева направо
/	Деление	
%	Модуль	Слева направо
+	Сложение	
-	Вычитание	
<<	Сдвиг влево	Слева направо
>>	Сдвиг вправо	

Оператор	Описание	Ассоциативность
<	Меньше чем	Слева направо
<=	Меньше чем или равно	
>	Больше чем	
=>	Больше чем или равно	Слева направо
==	Равенство	
!=	Неравенство	
&	Поразрядное AND	Слева направо
^	Поразрядное XOR	Слева направо
	Поразрядное OR	Слева направо
&&	Логическое AND	Слева направо
	Логическое OR	Слева направо
?:	Условный оператор	Справа налево
= *= /= %=	Операторы присваивания	Справа налево
+ = - =		
& = ^ =		
! = << =		
>> =		
,	Оператор запятая	Справа налево

Рассмотрим следующее выражение.

`b | c & d * e`

Оператор умножения имеет более высокий приоритет, чем оба поразрядных оператора OR и AND, поскольку в таблице А.5 он находится выше этих операторов. Аналогично, поразрядный оператор AND имеет более высокий приоритет, чем поразрядный оператор OR, поскольку последний находится двумя строчками ниже. Поэтому данное выражение будет вычислено следующим образом.

`b | ( c & ( d * e ) )`

Рассмотрим еще одно выражение.

`b % c * d`

Поскольку операторы модуля и перемножения находятся в одной группе таблицы А.5, они имеют одинаковый приоритет. Ассоциативность для этих операторов указана слева направо, что говорит о том, что выражение должно вычисляться следующим образом.

`( b % c ) * d`

Еще один пример выражения

```
++a->b
```

должен вычисляться следующим образом

```
++ (a->b)
```

поскольку оператор `->` имеет более высокий приоритет, чем оператор `++`.

Наконец, поскольку операторы присваивания находятся в одной группе и их ассоциативность реализуется справа налево, то утверждение

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

будет вычисляться как

```
a = (b = 0);
```

что приводит к последовательной установке значений переменных `a` и `b` в 0.

В случае выражения

```
x[i] + ++i
```

не понятно, в каком порядке компилятор будет производить вычисления, сначала будет вычислена правая сторона или же левая сторона от оператора `+`. В данном случае это важно, поскольку это влияет на общий результат вычисления всего выражения, т.к. если сначала будет выполнен инкремент переменной `i`, то будет использован другой элемент массива `x[i]`.

Еще одним примером неопределенной ситуации с порядком вычислений будет следующее выражение.

```
x[i] = ++i
```

Здесь нельзя точно сказать, когда будет инкрементирована переменная `i`: до или после того, как будет произведено обращение к элементу массива `x`.

Порядок вычисления аргументов функции также не определен. Поэтому при вызове функции

```
f (i, ++i);
```

нельзя точно сказать, будет ли первый аргумент `i` инкрементирован до того, как его значение будет передано в функцию.

В языке C гарантируется, что операторы `&&` и `||` будут вычисляться слева направо. Более того, в случае с оператором `&&` гарантируется, что второй операнд не будет вычисляться, если результат расчета первого операнда будет равен нулю. Также для оператора `||` гарантируется, что второй операнд не будет вычисляться, если результат расчета первого операнда не равен нулю.

Этот факт необходимо учитывать при написании следующих выражений.

```
if ( dataFlag || checkData (myData) )  
...
```

Поскольку в случае равенства переменной `dataFlag` нулю, не будет вызвана функция `checkData`. Рассмотрим еще один пример. Если массив `a` содержит `n` элементов, то следующее условное выражение

```
if (index >= 0 && index < n && a[index] == 0)
...

```

будет выполнено только в том случае, если значение индекса находится в заданных пределах и значение элемента равно нулю.

5.2. Константные выражения

Константными выражениями называются такие выражения, в которых каждый термин является константным значением. Константные выражения требуются в следующих ситуациях.

1. Как значение для выбора утверждений в конструкциях `switch`.
2. Для задания размеров массивов при инициализации или при глобальных объявлениях.
3. Для присваивания значения идентификаторам перечислимого типа.
4. Для задания размеров битовых полей при описании структур.
5. Для присваивания инициализирующих значений статическим переменным.
6. Для присваивания инициализирующих значений глобальным переменным.
7. Как выражения для утверждений препроцессора `#if`.

В первых четырех случаях константные выражения должны состоять из целочисленных констант, символьных констант, констант перечислимого типа и выражений `sizeof`. В качестве операторов могут использоваться арифметические операторы, поразрядные операторы, операторы отношений, условные операторы и приведение типов. Оператор `sizeof` нельзя использовать с массивами переменной длины, поскольку размер таких массивов задается во время выполнения программы и поэтому не является константным выражением.

В пятом и шестом случаях, в дополнение к правилам, перечисленным выше, можно явно или неявно использовать оператор адреса. Однако его можно использовать только с глобальными или статическими переменными. Поэтому, например, выражение

```
&x + 10
```

будет разрешенным константным выражением при условии, что переменная `x` является глобальной или статической переменной. Более того, выражение

```
&a[10] - 5
```

также будет разрешенным константным выражением, если `a` будет глобальным или статическим массивом. Наконец, поскольку `&a[0]` эквивалентно выражению `a`, то выражение

```
a + sizeof (char) * 100
```

также будет константным выражением.

В последнем случае, где требуется константное выражение для утверждения препроцессора `#if`, правила те же самые, как и в первых четырех случаях, за исключением того, что оператор `sizeof`, константы перечислимого типа и оператор приведения типа не могут быть использованы. Однако оператор `defined` разрешен (см. раздел 9.2.3).

5.3. Арифметические операторы

Полагая, что

`a, b` являются выражениями любого базового типа данных, за исключением `void`;

`i, j` являются выражениями любого целочисленного типа данных,

тогда выражение:

`-a` инвертирует значение `a`;

`+a` возвращает значение `a`;

`a + b` суммирует `a` и `b`;

`a - b` вычитает `b` из `a`;

`a * b` перемножает `a` и `b`;

`a / b` делит `a` на `b`;

`i % j` получает остаток от деления `i` на `j`.

В каждом случае выполняются обычные арифметические преобразования для всех операндов (см. раздел 5.17). Если `a` имеет беззнаковый тип, то `-a` вычисляется с помощью целочисленного расширения, вычитания значения из наибольшего значения для заданного типа и сложения результата с единицей.

Если делятся два целочисленных значения, то результат усекается. В случае одного отрицательного операнда, направление усекаения не определено (например, на некоторых компьютерах результат деления $-3/2$ может дать значение 1, а на других -2). В остальных случаях усекаение всегда производится в сторону нуля ($3/2$ всегда дает 1). Описание арифметических операций для указателей представлено в разделе 5.15.

5.4. Логические операторы

Полагая, что выражения `a` и `b` являются выражениями любого базового типа за исключением `void`, или что оба выражения являются указателями, тогда выражение:

`a && b` будет иметь значение 1, если ни выражение `a`, ни выражение `b` не равны нулю, и значение 0 — в противном случае (выражение `b` будет вычисляться только в том случае, если `a` не равно нулю);

`a || b` будет иметь значение 1, если оба выражения `a` и `b` не равны нулю, и значение 0 — в противном случае (выражение `b` будет вычисляться только в том случае, если `a` равно нулю);

`! a` будет иметь значение 1, если `a` равно нулю, и значение 0 — в противном случае.

Стандартные арифметические преобразования применимы к `a` и `b` (см. раздел 5.17). Типом результирующего значения во всех случаях будет `int`.

5.5 Операторы отношения

Полагая, что выражения *a* и *b* являются выражениями любого базового типа за исключением `void`, или оба выражения являются указателями, тогда:

<code>a < b</code>	имеет значение 1, если <i>a</i> меньше <i>b</i> , и 0 — в противном случае;
<code>a <= b</code>	имеет значение 1, если <i>a</i> меньше или равно <i>b</i> , и 0 — в противном случае;
<code>a > b</code>	имеет значение 1, если <i>a</i> больше чем <i>b</i> , и 0 — в противном случае;
<code>a >= b</code>	имеет значение 1, если <i>a</i> больше или равно <i>b</i> , и 0 — в противном случае;
<code>a == b</code>	имеет значение 1, если <i>a</i> равно <i>b</i> , и 0 — в противном случае;
<code>a != b</code>	имеет значение 1, если <i>a</i> не равно <i>b</i> , и 0 — в противном случае.

Стандартные арифметические преобразования применимы к *a* и *b* (см. раздел 5.17). Первые четыре оператора производят проверку для указателей только в том случае, если они оба указывают на один и тот же массив или на члены одной структуры или объединения. Типом результирующего значения во всех случаях будет `int`.

5.6. Поразрядные операторы

Предположим, что *i*, *j*, *n* являются выражениями любого целочисленного типа данных. Тогда выражения:

<code>i & j</code>	выполняют поразрядное И для <i>i</i> и <i>j</i> ;
<code>i j</code>	выполняют поразрядное ИЛИ для <i>i</i> и <i>j</i> ;
<code>i ^ j</code>	выполняют поразрядное исключающее ИЛИ для <i>i</i> и <i>j</i> ;
<code>~i</code>	выполняет одинарное дополнение для <i>i</i> ;
<code>i << n</code>	сдвигает <i>i</i> влево на <i>n</i> битов;
<code>i >> n</code>	сдвигает <i>i</i> вправо на <i>n</i> битов.

Стандартные арифметические преобразования применимы к операндам, за исключением операторов `<<` и `>>`, когда могут использоваться только целочисленные расширения (см. раздел 5.17).

Если количество сдвигов будет отрицательным или больше, чем количество битов, предназначенных для сдвига, то результат операции будет неопределенным. На некоторых компьютерах сдвиг вправо является арифметическим (производится заполнение знаком), на других — логическим (производится заполнение нулем). Тип результирующего значения задается левым операндом.

5.7. Операторы инкремента и декремента

Если *lv* является левосторонним модифицируемым выражением, тип которого не задан с ключевым словом `const`, тогда выражение:

<code>++lv</code>	инкрементирует значение <i>lv</i> и затем использует это значение в выражениях;
<code>lv++</code>	использует значение <i>lv</i> как значение для расчета выражения и затем инкрементирует его;

--lv	декрементирует значение lv и затем использует это значение в выражениях;
lv--	использует значение lv как значение для расчета выражения и затем декрементирует его.

В разделе 5.15 описаны эти операции для указателей.

5.8. Операторы присваивания

Если lv является левосторонним модифицируемым выражением, тип которого не задан с ключевым словом `const`, `op` является любой оператор, который может использоваться как оператор присваивания (см. табл. A.5), а а является выражением, тогда выражения:

lv = а	сохраняют значения а в lv;
lv op= а	применяют оператор <code>op</code> к lv и а и затем сохраняют результат в lv.

Если в первом выражении, а является одним из базовых типов данных (за исключением `void`), то он преобразовывается к типу lv. Если lv является указателем, то а тоже должен быть указателем и указывать на тот же тип, на тип `void` или быть нулевым указателем.

Если lv является указателем на тип `void`, когда а может быть указателем любого типа. Второе выражение трактуется как запись `lv = lv op (a)`, за исключением того, что lv вычисляется только один раз (`x[i++] += 10`).

5.9. Условные операторы

Если а, b и с являются выражениями, то выражение `a?b:c` будет возвращать значение b, если а не равно 0, в противном случае будет возвращено выражение с. Вычисляются только выражения b или с.

Выражения а и b должны возвращать один и тот же тип данных. Если же это условие не выполняется, но оба выражения возвращают арифметические типы данных, то автоматически выполняется приведение типов для получения одинаковых типов. Если одно из выражений возвращает указатель, а второе является нулем, то последнее трактуется как нулевой указатель на тот же тип данных, что и в первом случае. Если один из указателей указывает на тип `void`, а другой указывает на какой-либо тип, отличный от `void`, то последний преобразовывается как указатель на тип `void`, и это будет возвращаемым значением.

5.10. Оператор приведения типа

Предполагая, что type является именем основного типа данных, перечислимого типа, любого производного типа данных или типа, определенного с помощью утверждения `typedef`, а а является выражением, то тогда выражение `(type)а` преобразует выражение а в заданный тип.

5.11. Оператор sizeof

Предполагая, что type является именем основного типа, описанного выше, а а является выражением, то тогда выражение

`sizeof (type)`

возвращает количество битов , необходимых для хранения указанного типа данных. В выражение

`sizeof a`

возвращается количество байтов, требуемое для хранения результата, возвращаемого после вычисления выражения `a`.

Если типом `type` будет `char`, то результирующее значение будет равно 1. Если `a` является именем массива, размер которого явно или неявно задан при инициализации, то `sizeof a` возвращает количество байтов, требуемое для хранения массива.

Типом целочисленного значения, возвращаемого оператором `sizeof`, является тип `size_t`, который определен в стандартном заголовочном файле `stddef.h`.

Если `a` является массивом переменной длины, то оператор `sizeof` рассчитывается во время выполнения. В противном случае `a` вычисляется во время компиляции и его результат может использоваться в константных выражениях (см. раздел 5.2).

5.12. Оператор запятая

Предполагая, что `a`, `b` являются выражениями, можно написать выражение

`a, b`

которое будет вычисляться последовательно, т.е. сначала будет вычислено выражение `a`, а затем выражение `b`. Тип результирующего значения будет такой же, что и тип выражения `b`.

5.13. Основные операции с массивами

Предполагая, что:

`a` объявлен как массив с `n` элементами;

`i` является выражением любого целочисленного типа данных;

`v` является выражением,

то тогда выражение:

`a[0]` ссылается на первый элемент массива `a`;

`a[n - 1]` ссылается на последний элемент массива `a`;

`a[i]` ссылается на элемент массива `a` с номером `i`;

`a[i] = v` сохраняет значение `v` в элементе массива `a[i]`.

В любом случае, тип результирующего значения тот же, что и тип элементов, содержащихся в массиве `a`. Для ознакомления с операциями над указателями и массивами обратитесь к разделу 5.15.

5.14. Основные операции со структурами

Предполагая, что:

`x` является левосторонним модифицируемым выражением типа `struct s`;

`y` является выражением типа `struct s`;

`m` является именем одного из членов структуры `s`;

`v` является выражением,

тогда выражение:

<code>x</code>	ссылается на целую структуру и имеет тип <code>struct s</code> ;
<code>y.m</code>	ссылается на член <code>m</code> структуры <code>y</code> и имеет тип, объявленный для члена структуры <code>m</code> ;
<code>x.m = v</code>	присваивает значение <code>v</code> члену <code>m</code> структуры <code>x</code> и имеет тип, объявленный для члена структуры <code>m</code> ;
<code>x = y</code>	присваивает <code>y</code> переменной <code>x</code> и имеет тип <code>struct s</code> ;
<code>f (y)</code>	вызывает функцию <code>f</code> , передавая содержимое структуры <code>y</code> в качестве аргумента. Внутри функции формальный параметр должен быть объявлен как тип <code>struct s</code> ;
<code>return y;</code>	возвращает структуру <code>y</code> . Возвращаемым типом для функции должен быть тип <code>struct s</code> .

5.15. Основные операции с указателями

Предположим, что:

<code>x</code>	является левосторонним выражением типа <code>t</code> ;
<code>pt</code>	является левосторонним модифицируемым выражением типа "указатель на <code>t</code> ";
<code>v</code>	является выражением,

тогда выражение:

<code>&x</code>	возвратит указатель на <code>x</code> и будет иметь тип "указатель на <code>t</code> ";
<code>pt = &x</code>	присвоит <code>pt</code> значение указателя на <code>x</code> и его тип будет "указатель на <code>t</code> ";
<code>pt = 0</code>	присвоит указателю <code>pt</code> нулевое значение;
<code>pt == 0</code>	проверит на равенство нулю значение указателя <code>pt</code> ;
<code>*pt</code>	возвратит значение, на которое ссылается указатель <code>pt</code> и которое имеет тип <code>t</code> ;
<code>*pt = v</code>	сохранит значение типа <code>t</code> переменной <code>v</code> в том месте, на которое указывает указатель <code>pt</code> .

Указатели на массивы

Предположим, что:

<code>a</code>	— это массив элементов типа <code>t</code> ;
<code>pa1</code>	является модифицируемым левосторонним выражением типа "указатель на <code>t</code> ", который указывает на элементы массива <code>a</code> ;
<code>pa2</code>	является левосторонним выражением типа "указатель на <code>t</code> ", который указывает на элементы массива <code>a</code> или на предыдущий последний элемент массива <code>a</code> ;
<code>v</code>	является выражением;
<code>n</code>	является целочисленным выражением,

тогда выражение:

<code>a, &a, &a[0]</code>	возвращает указатель на первый элемент;
<code>&a[n]</code>	возвращает указатель на элемент с номером <code>n</code> и имеет тип "указатель на <code>t</code> ";
<code>*pa1</code>	возвращает значение элемента, на который указывает <code>pa1</code> , и имеет тип <code>t</code> ;
<code>*pa1 = v</code>	сохраняет значение переменной <code>v</code> в элементе, на который указывает <code>pa1</code> и имеет тип <code>t</code> ;
<code>++pa1</code>	устанавливает указатель <code>pa1</code> на следующий элемент массива <code>a</code> , независимо от того, какого типа элементы содержатся в массиве <code>a</code> , и имеет тип "указатель на <code>t</code> ";
<code>--pa1</code>	устанавливает указатель <code>pa1</code> на предыдущий элемент массива <code>a</code> , независимо от того, какого типа элементы содержатся в массиве <code>a</code> , и имеет тип "указатель на <code>t</code> ";
<code>*++pa1</code>	инкрементирует значение указателя <code>pa1</code> и ссылается на значение (возвращает значение), на которое будет указывать <code>pa2</code> . Имеет тип <code>t</code> ;
<code>*pa1++</code>	возвращает значение, на которое указывает <code>pa1</code> , после чего инкрементирует значение указателя <code>pa1</code> . Имеет тип <code>t</code> ;
<code>pa1 + n</code>	создает указатель, который указывает на элемент, отстоящий на <code>n</code> элементов от того элемента, на который указывает <code>pa1</code> , и имеет тип "указатель на <code>t</code> ";
<code>pa1 - n</code>	создает указатель, который указывает на элемент, находящийся ближе на <code>n</code> элементов от того элемента, на который указывает <code>pa1</code> , и имеет тип "указатель на <code>t</code> ";
<code>*(pa1 + n) = v</code>	сохраняет значение переменной <code>v</code> в элементе, на который ссылается указатель <code>pa1 + n</code> и имеет тип "указатель на <code>t</code> ";
<code>pa1 < pa2</code>	производит проверку на предмет того, указывает ли указатель <code>pa1</code> на более ранний элемент, чем указатель <code>pa2</code> , и имеет тип <code>int</code> (все операторы отношения могут использоваться для сравнения двух указателей);
<code>pa2 - pa1</code>	возвращает количество элементов, находящихся между указателями <code>pa2</code> и <code>pa1</code> (предполагается, что <code>pa2</code> указывает на более дальний элемент, чем <code>pa1</code>), и имеет целочисленный тип;
<code>a + n</code>	возвращает указатель на элемент с номером <code>n</code> и имеет тип "указатель на <code>t</code> ". Упрощенная запись для выражения <code>&a[n]</code> ;
<code>*(a + n)</code>	возвращает значение элемента с номером <code>n</code> и имеет тип <code>t</code> . Аналогично записи <code>a[n]</code> .

Действительным типом для целочисленных значений, получаемый при вычитании двух указателей, является тип `ptrdiff_t`, который определен в стандартном заголовочном файле `stddef.h`.

Указатели на структуры

Предположим, что:

<code>x</code>	является левосторонним выражением типа <code>struct s</code> ;
<code>ps</code>	является модифицируемым левосторонним выражением типа "указатель на <code>struct s</code> ";
<code>m</code>	является именем члена структуры <code>s</code> и имеет тип <code>t</code> ;
<code>v</code>	является выражением,

тогда выражение:

<code>&x</code>	возвращает указатель на <code>x</code> и имеет тип "указатель на <code>struct s</code> ";
<code>ps = &x</code>	приводит к тому, что указатель <code>ps</code> указывает на <code>x</code> и имеет тип "указатель на <code>struct s</code> ";
<code>ps->m</code>	ссылается на член <code>m</code> структуры, на которую указывает указатель <code>ps</code> , и имеет тип <code>t</code> ;
<code>(*ps).m</code>	также ссылается на член <code>m</code> и, в любом случае, эквивалентен выражению <code>ps->m</code> ;
<code>ps->m = v</code>	сохраняет значение переменной <code>v</code> в члене <code>m</code> структуры, на которую указывает указатель <code>ps</code> , и имеет тип <code>t</code> .

5.16. Составные литералы

Составные литералы представляют имя типа, заключенное в круглые скобки, со следующим за ним списком инициализаторов. При этом создается значение заданного типа без имени, которое имеет область видимости, ограниченную блоком, в котором создано это значение, или глобальную видимость, если оно определено за пределами всех функций. В последнем случае все инициализаторы должны быть константными выражениями. Например, запись

```
(struct point) {.x = 0, .y = 0}
```

является выражением, которое возвращает структуру типа `struct point`, инициализированную заданными значениями. Значение этой структуры можно присвоить другой структуре того же типа

```
origin = (struct point) {.x = 0, .y = 0};
```

или передать в функцию в качестве аргумента типа `struct point`, например, следующим образом.

```
moveToPoint ((struct point) {.x = 0, .y = 0});
```

Так можно определять не только структуры, но и другие типы, на пример, если переменная `intPtr` имеет тип `int *`, то утверждение

```
intPtr = (int [100]) {[0] = 1, [50] = 50, [99] = 99};
```

которое может находиться в любом месте программы, приводит к тому, что `intPtr` будет указывать на массив из 100 элементов, первые три элемента которого инициализированы заданными значениями.

Если размер массива не задан, то он определяется по списку инициализаторов.

5.17. Преобразования базовых типов данных

В языке программирования C может происходить преобразование операндов в арифметических выражениях, которое называют *стандартным арифметическим преобразованием*.

- Шаг 1.** Если один из операндов выражения имеет тип `long double`, то все другие операнды приводятся к типу `long double` и результирующее значение имеет этот тип.
- Шаг 2.** Если один из операндов выражения имеет тип `double`, то все другие операнды приводятся к типу `double` и результирующее значение имеет этот тип.
- Шаг 3.** Если один из операндов выражения имеет тип `float`, то все другие операнды приводятся к типу `float` и результирующее значение имеет этот тип.
- Шаг 4.** Если один из операндов выражения имеет тип `_Bool`, `char`, `short int`, `int`, битовое поле или перечислимый тип, то они приводятся к типу `int`, но при условии, что этот тип может полностью перекрыть весь диапазон значений. В противном случае происходит приведение к типу `unsigned int`. Если оба операнда имеют один и тот же тип, то он и будет типом результата.
- Шаг 5.** Если оба операнда являются знаковыми или беззнаковыми, то наименьший целочисленный тип приводится к большему целочисленному типу, который и будет представлять результирующее значение.
- Шаг 6.** Если беззнаковый операнд равен или больше по размеру, чем знаковый операнд, тогда знаковый операнд приводится к типу беззнакового операнда, который и будет типом результата.
- Шаг 7.** Если знаковый операнд может включать все значения беззнакового операнда, то последний преобразовывается к типу первого, который и будет типом результата.
- Шаг 8.** Если достигнут этот шаг, то оба операнда преобразовываются в беззнаковый тип в соответствии с аналогичным знаковым типом.

Шаг 4 обычно формально называют *целочисленным расширением*.

Преобразование операндов в большинстве случаев работает корректно, хотя необходимо учитывать следующие моменты.

1. Преобразование типа `char` к типу `int` на некоторых машинах может привести к типу со знаком, если тип `char` объявлен не как `unsigned`.
2. Преобразование целочисленного значения со знаком в большее целочисленное значение со знаком приводит к заполнению знаком битов слева, а преобразование целочисленного значения без знака в большее целочисленное значение приводит к заполнению нулями битов слева.
3. Преобразование любого значения в тип `_Bool` приводит к нулевому результату, если значение равно нулю, и к единице в противном случае.
4. Преобразование большего целочисленного значения в меньшее приводит к усечению значения слева.
5. Преобразование вещественного значения в целочисленное приводит к усечению десятичной части значения. Если размер целочисленного значения

недостаточен для размещения вещественного числа, результат будет неопределенным, как и результат преобразования отрицательного вещественного числа в беззнаковое целое значение.

6. Преобразование вещественного числа в меньшее число приводит к усечению значения после округления.

6.0. Классы памяти и область видимости

Термин *класс памяти* относится как к способу, с помощью которого память распределяется компилятором для переменных, так и к области видимости для отдельного описания функции. Классы памяти могут называться `auto`, `static`, `extern` и `register`. Если класс памяти не задан при объявлении, то класс памяти присваивается по умолчанию, о чем будет сказано в дальнейшем.

Термин *область видимости* относится к непрерывной области памяти, из которой можно обратиться к отдельному идентификатору в программе. Если идентификатор описан за пределами всех функций или блоков утверждений (в дальнейшем мы будем ссылаться на него как на БЛОК), то к нему можно обратиться из любого места программы. Идентификаторы, описанные в БЛОКЕ, являются локальными для этого БЛОКА, и причем аналогичный идентификатор можно объявлять за пределами БЛОКА.

Имена меток видны только в пределах БЛОКА, как и имена формальных параметров. Метки, структуры члены структур, объединения, члены объединений и идентификаторы перечислимых типов формально не отличаются друг от друга, как и имена функций и переменных. Однако идентификаторы перечислимых типов отличаются от имен переменных и от идентификаторов других перечислимых типов в одной и той же области видимости.

6.1. Функции

Если при описании функции задается класс памяти, то это должен быть или `static`, или `extern`. На функции, которые объявляются с ключевым словом `static`, можно ссылаться только из того файла, который содержит данную функцию. Функции, которые объявлены с ключевым словом `extern` (или вообще не имеют спецификации памяти), могут быть вызваны функциями из других файлов.

6.2. Переменные

В табл. А.6 представлены различные классы памяти, которые можно использовать при объявлении переменных, а также их области видимости и методы инициализации.

Таблица А.6. Переменные: классы памяти, области видимости и методы инициализации

Класс памяти	Место объявления	Место ссылки	Чем инициализируется	Комментарий
static	За пределами БЛОКА	Из любого места файла	Только константным выражением	Инициализируется один раз при запуске программы, значение сохраняется в блоке, значение по умолчанию – 0
	Внутри БЛОКА	В пределах БЛОКА		
extern	За пределами БЛОКА	Из любого места файла	Только константным выражением	Переменная может быть объявлена без ключевого слова extern или со словом extern и инициализацией
	Внутри БЛОКА	В пределах БЛОКА		
auto	Внутри БЛОКА	В пределах БЛОКА	Любым допустимым выражением	Инициализируется каждый раз при входе в БЛОК, нет значения по умолчанию
register	Внутри БЛОКА	В пределах БЛОКА	Любым допустимым выражением	Присваивание значения регистру не гарантируется, ограничения на объявления различных типов, нельзя использовать адрес регистровой переменной, инициализируется при каждом входе в БЛОК, нет значения по умолчанию
omitted	За пределами БЛОКА	Из любого места файла или из других файлов, которые содержат соответствующие объявления	Только константным выражением	Инициализируется при запуске программы Значение по умолчанию – 0
	Внутри БЛОКА	См. auto	См. auto	

7.0. Функции

В этом разделе обобщены сведения о синтаксисе и операциях над функциями.

7.1. Объявление функций

Общий формат описания функции при ее объявлении следующий.

```
returnType name ( type1 param1, type2 param2, ... )
{
    variableDeclarations
    programStatement
    programStatement
    ...
    return expression;
}
```

Здесь описана функция с именем `name`, которая возвращает значение типа `returnType` и которая принимает формальные параметры `param1`, `param2`, ... Параметр `param1` объявлен как тип `type1`, параметр `param2` объявлен как тип `type2` и т.д.

Локальные переменные обычно объявляются в начале функции, хотя это не обязательно. Они могут объявляться в любом месте, но в этом случае к ним можно обращаться только из утверждений, расположенных в функции после объявления переменной.

Если функция не должна возвращать значение, то она должна быть объявлена с ключевым словом `void`.

Если слово `void` помещено в круглые скобки, то функция не принимает никаких аргументов. Если в конце списка параметров находится многоточие (...), или присутствуют только многоточие без параметров, то функция принимает переменное число аргументов.

```
int printf (char *format, ...)
{
    ...
}
```

При объявлении одномерного массива не нужно задавать количество элементов массива. Для многомерного массива указывается размер каждого измерения, за исключением первого. Для более полного ознакомления с ключевым словом `return` обратитесь к разделу 8.9.

Как напоминание компилятору, в начале объявления функции может находиться ключевое слово `inline`. Некоторые компиляторы вместо вызова функции непосредственно подставляют код функции в программу, что приводит к более быстрому ее выполнению.

```
inline int min (int a, int b)
{
    return ( a < b ? a : b);
}
```

7.2. Вызов функции

Общий формат записи утверждения для вызова функции следующий.

```
name ( arg1, arg2, ... )
```

В этом утверждении вызывается функция с именем `name`, в которую в качестве аргументов передаются значения `arg1, arg2, ...`. Если функция не принимает аргументов, то записываются только круглые скобки (например, `initialize()`)

Если вы обращаетесь к функции, которая объявлена после вашего обращения или находится в другом файле, то необходимо создать *прототип объявления* для функции, который имеет следующий формат.

```
returnType name (type1 param1, type2 param2, ... );
```

В этом прототипе объявления для компилятора указываются тип возвращаемого значения, количество принимаемых аргументов и тип каждого аргумента. Например, в строке

```
long double power (double x, int n);
```

говорится о том, что функция с именем `power` возвращает тип `long double` и принимает два аргумента: первый типа `double` и второй типа `int`. Имена аргументов в круглых скобках являются фиктивными и могут быть пропущены, поэтому утверждение

```
long double power (double, int);
```

полностью аналогично предыдущему.

Если компилятор считал объявление функции, то после этого он имеет представление о типах всех аргументов и при вызове функции будет автоматически преобразовывать все значения (если это возможно) к заданному типу.

Если до вызова функции компилятор не встретил объявление функции или ее прототипа, то компилятор будет считать типом возвращаемого значения тип `int`, автоматически преобразует все аргументы типа `float` к типу `double` и выполнит преобразования целочисленных аргументов в соответствии с тем, как описано в разделе 5.17. Аргументы другого типа передаются без преобразований.

Функции, которые принимают переменное количество аргументов, также предварительно должны быть объявлены. В противном случае компилятор будет считать, что функция принимает фиксированное количество аргументов, которое реально использовано при вызове.

Функции, в которых для возвращаемого типа использовано ключевое слово `void`, будут проверяться компилятором на предмет некорректного использования возвращаемого типа.

Все аргументы передаются в функцию по значению и поэтому не могут быть изменены. Если в функцию передается указатель, то функция может изменить значение, на которое ссылается указатель, но не может изменить значение самого указателя.

7.3. Указатели на функции

Само имя функции без круглых скобок является указателем на функцию. Оператор адреса может использоваться с именем функции для получения отдельного указателя на нее. Если идентификатор `fp` является указателем на функцию, то соответствующая функция может быть вызвана с помощью утверждения

```
fp ()  
или  
(*fp) ()
```

Если функция принимает аргументы, то они должны быть перечислены в круглых скобках.

8.0. Утверждения

Утверждениями в программе являются любые допустимые выражения (обычно это присваивания или вызовы функций), за которыми непосредственно стоит точка с запятой. Специальные виды утверждений описаны в следующем разделе. Перед любым утверждением может находиться метка, которая представляет идентификатор, заканчивающийся двоеточием (см. раздел 8.6).

8.1. Составные утверждения

Несколько утверждений, заключенных в фигурные скобки, называются *составным утверждением* или *блоком*. Составное утверждение может находиться там же, где разрешено ставить обычное одинарное утверждение. В блоке можно объявлять внутренние переменные, которые будут подменять все аналогичные имена, объявленные за пределами блока. Область видимости таких переменных ограничена блоком.

8.2. Утверждение break

Общий формат объявления утверждения break следующий.

```
break;
```

Размещение утверждения break в циклах for, while, do или утверждениях switch приводит к тому, что при достижении этого утверждения выполнение цикла или утверждения switch будет закончено, а выполнение программы будет продолжено с утверждения, которое непосредственно следует за циклом.

8.3. Утверждение continue

Общий формат объявления утверждения continue следующий.

```
continue;
```

Размещение утверждения continue в циклах приводит к тому, что при достижении этого утверждения все дальнейшие утверждения до конца цикла будут пропущены и выполнение цикла будет продолжено с самого начала цикла.

8.4. Цикл do

Общий формат объявления цикла do следующий.

```
do  
    programStatement  
while ( expression );
```

Тело цикла составляют утверждения programStatement, которые выполняются до тех пор, пока результат расчета выражения expression не будет равен нулю. Обратите

внимание, что поскольку выражение `expression` рассчитывается после выполнения утверждений `programStatement`, то в этом цикле гарантируется, что утверждения `programStatement` будут выполнены по крайней мере один раз.

8.5. Цикл `for`

Общий формат объявления цикла `for` следующий.

```
for ( expression_1; expression_2; expression_3 )
    programStatement
```

При этом выражение `expression_1` рассчитывается еще до того, как начнется выполнение цикла. Затем рассчитывается выражение `expression_2`. Если значение этого выражения не равно нулю, то выполняются утверждения тела цикла `programStatement` и затем рассчитывается выражение `expression_3`. Выполнение тела цикла `programStatement` и расчет выражения `expression_3` продолжают до тех пор, пока значение выражения `expression_2` не станет равным нулю.

Обратите внимание, что поскольку выражение `expression_2` рассчитывается каждый раз перед выполнением тела цикла, то тело цикла `programStatement` не будет выполнено ни разу, если при входе в цикл значение выражения `expression_2` равно 0.

Локальные переменные для цикла `for` могут быть объявлены в выражении `expression_1`. Областью видимости таких переменных является тело цикла. Например, в цикле

```
for ( int i = 0; i < 100; ++i)
    ...
```

объявляется целочисленная переменная `i`, которая инициализируется значением 0 при входе в цикл. Эта переменная может использоваться в утверждениях в теле цикла, но не за пределами цикла.

8.6. Утверждение `goto`

Общий формат объявления утверждения `goto` следующий.

```
goto identifier;
```

Выполнение утверждения `goto` приводит в передаче управления тому утверждению, перед которым находится метка `identifier`. Утверждение с этой меткой должно находиться в той же самой функции, где и соответствующее утверждение `goto`.

8.7. Условие `if`

Первый формат объявления утверждения `if` следующий.

```
if ( expression )
    programStatement
```

Если результат расчета выражения `expression` не равен нулю, то выполняется тело утверждения `programStatement`, в противном случае выполнение тела утверждения пропускается.

Второй формат объявления утверждения `if` следующий.

```
if ( expression )
    programStatement_1
```

```
else
    programStatement_2
```

Если выражение `expression` не равно нулю, то выполняются утверждения из `programStatement_1`, в противном случае выполняются утверждения `programStatement_2`. Если утверждения `programStatement_2` являются еще одним условием `if`, то будет реализована цепочка `if-else if`.

```
if ( expression_1 )
    programStatement_1
else if ( expression_2 )
    programStatement_2
```

```
...
else
    programStatement_n
```

Ключевое слово `else` всегда связывается с последним ключевым словом `if`, которое не содержит `else`. С помощью фигурных скобок можно изменить взаимосвязи этих слов.

8.8. Утверждение `null`

Общий формат объявления утверждения `null` следующий.

```
;
```

Выполнение утверждения `null` не дает никакого эффекта и используется в основном для соблюдения синтаксических правил в циклах. Например, в следующем цикле производится копирование символов строки, на которую ссылается указатель `from`, в строку, на которую ссылается указатель `to`.

```
while ( *to++ = *from++ )
    ;
```

Здесь утверждение `null` используется для того, чтобы соблюсти требование о необходимости размещения утверждения после закрывающей круглой скобки в цикле `while`.

8.9. Утверждение `return`

Первый формат объявления утверждения `return` следующий.

```
return;
```

Выполнение утверждения `return` приводит к тому, что выполнение программы передается непосредственно в вызывающую функцию. Такой формат используется только для возврата из функции, для которой не объявлено возвращаемое значение.

Если выполнение тела функции закончено и утверждение `return` не встретилось, то происходит возврат из функции таким образом, как будто утверждение `return` находится в самом конце последовательности утверждений. При этом никакое значение не возвращается.

Второй формат объявления утверждения `return` следующий.

```
return expression;
```

При этом значение выражения `expression` возвращается в вызывающую функцию. Если тип, полученный в результате расчета выражения `expression`, не соответствует типу, указанному при объявлении функции, то его значение автоматически преобразовывается к объявленному типу до возвращения.

8.10. Утверждение `switch`

Общий формат объявления утверждения `return` следующий.

```
switch ( expression )
{
    case constant_1:
        programStatement
        programStatement
        ...
        break;
    case constant_2:
        programStatement
        programStatement
        ...
        break;
    ...
    case constant_n:

        programStatement
        programStatement
        ...
        break;
    default:
        programStatement
        programStatement
        ...
        break;
}
```

Здесь рассчитывается выражение `expression` и результат расчета последовательно сравнивается со значениями константных выражений `constant_1`, `constant_2`, ..., `constant_n`. Если значение выражения `expression` совпадает с одним из этих значений, то выполняются утверждения, следующие непосредственно за этим значением. Если совпадений не встретилось, то выполняются утверждения, следующие за меткой `default`, если она поставлена. Если утверждений с меткой `default` нет в утверждении `switch`, то в случае отсутствия совпадений ничего не выполняется.

Результатом расчета выражения `expression` должны быть целочисленные типы, значения которых не должны совпадать. Пропуск утверждения `break` в одном из фрагментов приводит в продолжению выполнения утверждений на следующие фрагменты до тех пор, пока не встретится утверждение `break`.

8.11. Цикл while

Общий формат объявления утверждения while следующий.

```
while ( expression )  
    programStatement
```

В этом случае утверждения programStatement выполняются до тех пор, пока значение выражения expression не равно нулю. Обратите внимание, что поскольку выражение expression рассчитывается до того, как будут выполнены утверждения programStatement, то утверждения programStatement в некоторых случаях могут ни разу не выполниться.

9.0. Препроцессор

Препроцессор анализирует исходный файл до того, как к обработке кода приступит компилятор. Препроцессор выполняет следующее.

1. Заменяет триграфные последовательности (см. раздел 9.1) эквивалентными им конструкциями.
2. Объединяет все строки, которые заканчиваются символом обратной черты.(\) в единую строку.
3. Преобразовывает программу в поток лексем.
4. Удаляет комментарии, заменяя их пробелом.
5. Обрабатывает директивы препроцессора (смотри раздел 9.2) и расширяет макроопределения.

9.1. Триграфные последовательности

При работе с набором символов, не являющихся набором ASCII, используются последовательности из трех символов, называемые *триграфами*. В табл. А.7 перечислены все последовательности, которые распознаются и трактуются особым образом, когда они встречаются в коде программы или внутри символьной строки.

Таблица А.7. Триграфные последовательности

Триграф	Значение
??=	#
??(	[
??)	]
??<	{
??>	}
??/	\
??'	^
??!	
??-	~

9.2. Директивы препроцессора

Все директивы препроцессора начинаются символом `#`, который всегда должен быть первым символом, не считая пробела. Символ `#` может сопровождаться одним или несколькими символами `<Space>` или `<Tab>`.

9.2.1. Директива `#define`

Первый формат объявления директивы `#define` следующий.

```
#define name text
```

В этом случае объявляется идентификатор `name` и связывается с тем, что отображается после пробела за идентификатором `name` и до конца строки (`text`). Дальнейшее использование в программе идентификатора `name` приводит к тому, что все вхождения этого идентификатора в программе будут заменены на то, что скрывается под словом `text`.

Второй формат объявления директивы `#define` следующий.

```
#define name(param_1, param_2, ..., param_n) text
```

Здесь задается макроопределение с именем `name`, которое принимает аргументы в соответствии с заданными параметрами `param_1, param_2, ..., param_n`, каждый из которых является идентификатором. Дальнейшее использование имени `name` в программе приводит к тому, что будут заменены все вхождения этого имени вместе со списком аргументов на то, что скрывается под словом `text`.

Если макроопределение принимает различное количество аргументов, то в конце списка аргументов ставятся три точки. На те аргументы, которые не входят в перечень, можно сослаться с помощью специального идентификатора `__VA_ARGS__`. Например, в следующем объявлении макроопределения с именем `myPrintf` в качестве аргументов задается строка форматирования с некоторым количеством дополнительных аргументов:

```
#define myPrintf(...) printf ("DEBUG: " __VA_ARGS__);
```

Корректно это макроопределение можно использовать как

```
myPrintf ("Hello world!\n");
```

или как

```
myPrintf ("i = %i, j = %i\n", i, j);
```

Если описание требует более одной строки, то оно может продолжаться на следующую строку, причем предыдущая строка должна заканчиваться обратной чертой (`\`). После того как имя будет задано, в дальнейшем оно может использоваться в любом месте программы.

В описании для директивы `#define` можно использовать оператор `#`, за которым должно находиться имя аргумента. Во время компиляции препроцессор разместит двойные кавычки вокруг действительного значения, передаваемого в макроопределение. Таким образом, это имя будет преобразовано в символьную строку. Например, описание

```
#define printint(x) printf (# x " = %d\n", x)
```

при следующем вызове

```
printint (count);
```

будет преобразовано препроцессором в

```
printf ("count" " = %i\n", count);
```

что эквивалентно следующему.

```
printf ("count = %i\n", count);
```

Препроцессор разместит символ обратной черты (\) перед любым символом " или \, когда будет выполнять операции со строками. Поэтому объявление

```
#define str(x) # x
```

при вызове

```
str (The string "\t" contains a tab)
```

будет преобразовано в следующее.

```
"The string "\\t" contains a tab"
```

При использовании директивы #define также разрешено применять оператор ##, за которым должен располагаться аргумент макроопределения. Препроцессор принимает значение, которое передается в макроопределение при вызове, и создает отдельную лексему из аргумента макроопределения и лексемы, которая следует (или предв-
редв-яет) это значение. Например, при объявлении макроопределения

```
#define printx(n) printf ("%i\n", x ## n );
```

при вызове

```
printx (5)
```

будет получено следующее.

```
printf ("%i\n", x5);
```

Объявляя

```
#define printx(n) printf ("x" # n " = %i\n", x ## n );
```

при вызове

```
printx(10)
```

получим следующее.

```
printf ("x10 = %i\n", x10);
```

При этом были выполнены подстановка и объединение строк. Пробелы вокруг операторов # и ## не обязательны.

9.2.2. Директива `#error`

Общий формат объявления директивы `#define` следующий.

```
#error text
...
```

Текст, заданный словом `text` будет выведен препроцессором как сообщение об ошибке.

9.2.3 Директива `#if`

Первый формат объявления директивы `#if` следующий.

```
#if constant_expression
...
#endif
```

При этом сначала рассчитывается значение константного выражения `constant_expression`. Если полученный результат не равен нулю, выполняются все строки программы вплоть до директивы `#endif`. В противном случае все эти строки пропускаются и не обрабатываются ни препроцессором, ни компилятором.

Второй формат объявления директивы `#if` следующий.

```
#if constant_expression_1
...
#elif constant_expression_2
...
#elif constant_expression_n
...
#else
...
#endif
```

При этом сначала рассчитывается значение константного выражения `constant_expression_1`. Если полученный результат не равен нулю, выполняются все строки программы вплоть до директивы `#elif`. В противном случае, если результат расчета константного выражения `constant_expression_2` не равен нулю, выполняются все строки программы вплоть до следующей директивы `#elif`, а оставшиеся до директивы `#endif` строки пропускаются. Если результат расчета всех константных выражений равен нулю, то выполняются строки после директивы `#else`, если она включена. Может использоваться специальный оператор `defined` как часть константного выражения, поэтому строки

```
#if defined (DEBUG)
...
#endif
```

приведут к тому, что код между директивами `#if` и `#endif` будет обрабатываться только в том случае, если идентификатор `DEBUG` был предварительно объявлен (см. раздел 9.2.4). Крутые скобки вокруг идентификатора не являются обязательными, поэтому утверждение

```
#if defined DEBUG
```

выполняется точно так же.

9.2.4. Директива `#ifdef`

Общий формат объявления директивы `#ifdef` следующий.

```
#ifdef identifier
...
#endif
```

Если идентификатор `identifier` был предварительно объявлен (либо с помощью директивы `#define`, либо в командной строке с использованием опции `-D`), будут обработаны все строки программы, вплоть до директивы `#endif`, в противном случае они будут пропущены. Как и для директивы `#if`, можно использовать директивы `#elif`, `#ifdef`, и `#else`.

9.2.5. Директива `#ifndef`

Общий формат объявления директивы `#ifndef` следующий.

```
#ifndef identifier
...
#endif
```

Если идентификатор `identifier` не был предварительно объявлен (либо с помощью директивы `#define`, либо в командной строке с использованием опции `-D`), будут обработаны все строки программы, вплоть до директивы `#endif`, в противном случае они будут пропущены. Как и для директивы `#if`, можно использовать директивы `#elif`, `#ifdef`, и `#else`.

9.2.6. Директива `#include`

Первый формат объявления директивы `#include` следующий.

```
#include "fileName"
```

При этом препроцессор выполняет поиск файла `fileName` в определенном при реализации каталоге или каталогах. Обычно поиск сначала выполняется в каталоге, который содержит исходный файл. Если файл не обнаружен, то другие каталоги, в которых будет производиться поиск, зависят от реализации языка и платформы. После того как файл будет найден, содержимое файла включается в программу в том месте, где встретилась директива `#include`. Директивы препроцессора, находящиеся в подключаемых файлах, обрабатываются препроцессором, поэтому в этом файле могут также находиться директивы `#include`.

Второй формат объявления директивы `#include` следующий.

```
#include <fileName>
```

В данном случае препроцессор ищет заданный файл только в стандартных каталогах. Действия, которые выполняются после определения файла, аналогичны описанному выше. Для требуемого имени файла можно использовать предварительно определенное имя, и при этом произойдет его обработка. Например, следующие строки будут полностью корректны.

```
#define DATABASE_DEFS </usr/data/database.h>
...
#include DATABASE_DEFS
```

9.2.7. Директива `#line`

Общий формат объявления директивы `#line` следующий.

```
#line constant "fileName"
```

Эта директива заставляет компилятор трактовать последовательность строк программы так, как будто это строки исходного файла `fileName`. Если имя файла не задано, то используется ранее указанное имя файла для предыдущей директивы `#line` или имя исходного файла (если никакого имени ранее не было определено).

Директива `#line` в основном используется для контроля имени файла и номеров строк, которые отображаются при получении сообщений об ошибках времени компиляции.

9.2.8. Директива `#pragma`

Общий формат объявления директивы `#pragma` следующий.

```
#pragma text
```

Такая директива заставляет препроцессор выполнять некоторые действия, которые зависят от реализации компилятора. Так, директива

```
#pragma loop_opt(on)
```

может привести к тому, что некоторые компиляторы будут дополнительно производить оптимизацию циклов. Если такую директиву встретит компилятор, который не распознает указание `loop_opt`, то оно будет просто проигнорировано.

Специальное ключевое слово `STDC` используется после директивы `#pragma` для определенных целей. Стандартно поддерживаемые переключатели, которые могут встретиться после `#pragma STDC`, — это `FP_CONTRACT`, `FENV_ACCESS` и `CX_LIMITED_RANGE`.

9.2.9. Директива `#undef`

Общий формат объявления директивы `#undef` следующий

```
#undef identifier
```

При этом указанный идентификатор `identifier` становится неопределенным. Последующая реакция директив `#ifdef` или `#ifndef` будет такая, как будто идентификатор никогда не был определен.

9.2.10. Директива `#`

Это ничего не значащая директива, которая игнорируется препроцессором.

9.3. Предопределенные идентификаторы

Все идентификаторы, перечисленные в табл. А.8, распознаются препроцессором.

Таблица А.8. Предопределенные идентификаторы

Идентификатор	Значение
__LINE__	Номер компилируемой строки
__FILE__	Имя исходного компилируемого файла
__DATE__	Дамп даты компилируемого файла в формате "mm dd yyyy"
__TIME__	Дамп времени компилируемого файла в формате "hh:mm:ss"
__STDC__	Соответствует 1, если компилятор поддерживает стандарт ANSI, в противном случае — 0
__STDC_HOSTED__	Соответствует 1, если это основная реализация, в противном случае — 0
__STDC_VERSION__	Соответствует 199901L

Б

Стандартные библиотеки языка C

Стандартные библиотеки языка C# содержат большое число функций, которые можно использовать в программах. В этом разделе перечислены далеко не все подобные функции, а только наиболее часто используемые. Для знакомства со всеми доступными функциями следует прочесть документацию, которая прилагается к компилятору, а также внимательно изучить приложение Д, “Ресурсы”.

Среди стандартных подпрограмм, которые не описаны в данном разделе, есть функции для работы с датой и временем (такие как `time`, `ctime` и `localtime`), выполнения удаленных переходов (`setjmp` и `longjmp`), проведения диагностики (`assert`), обработки различных аргументов (`va_list`, `va_start`, `va_arg`, и `va_end`), создания сигналов (`signal` и `raise`), локализации (в соответствии с файлом `locale.h`) и работы с расширенными символьными строками.

Стандартные заголовочные файлы

В этом разделе описывается содержимое некоторых стандартных подключаемых файлов: `stddef.h`, `stdbool.h`, `limits.h`, `float.h` и `stdinit.h`.

`stddef.h`

Этот заголовочный файл содержит определение отдельных стандартных идентификаторов, которые приведены ниже.

Идентификатор	Значение
<code>NULL</code>	Указатель на нулевую константу
<code>offsetof</code> (<code>structure</code> , <code>member</code>)	Смещение в байтах члена структуры от начала структуры, тип результата <code>size_t</code>

Идентификатор	Значение
<code>ptrdiff_t</code>	Тип целочисленного значения, полученный при вычитании двух указателей
<code>size_t</code>	Тип целочисленного значения, полученный при вычислении размера с помощью оператора <code>wchar_t</code> . Тип целочисленного значения, необходимый при работе с расширенными символами (см. Приложение А, “Справочник по языку C),”)

limits.h

Этот заголовочный файл содержит различные, зависящие от реализации, символьные и целочисленные типы данных. Указанные минимальные значения гарантируются стандартом ANSI. Значения указываются в круглых скобках после описания.

Идентификатор	Значение
<code>CHAR_BIT</code>	Количество битов в типе <code>char</code> (8)
<code>CHAR_MAX</code>	Максимальное значение для объектов типа <code>char</code> (127 при использовании значений со знаком и 255 в противном случае)
<code>CHAR_MIN</code>	Минимальное значение для объектов типа <code>char</code> (-127 при использовании значений со знаком и 0 в противном случае)
<code>SCHAR_MAX</code>	Максимальное значение для объектов типа <code>signed char</code> (127)
<code>SCHAR_MIN</code>	Минимальное значение для объектов типа <code>signed char</code> (-127)
<code>UCHAR_MAX</code>	Максимальное значение для объектов типа <code>unsigned char</code> (255)
<code>SHRT_MAX</code>	Максимальное значение для объектов типа <code>short int</code> (32767)
<code>SHRT_MIN</code>	Минимальное значение для объектов типа <code>short int</code> (-32767)
<code>USHRT_MAX</code>	Максимальное значение для объектов типа <code>unsigned short int</code> (65535)
<code>INT_MAX</code>	Максимальное значение для объектов типа <code>int</code> (32767)
<code>INT_MIN</code>	Минимальное значение для объектов типа <code>int</code> (-32767)
<code>UINT_MAX</code>	Максимальное значение для объектов типа <code>unsigned int</code> (65535)
<code>LONG_MAX</code>	Максимальное значение для объектов типа <code>long int</code> (2147483647)
<code>LONG_MIN</code>	Минимальное значение для объектов типа <code>long int</code> (-2147483647)
<code>ULONG_MAX</code>	Максимальное значение для объектов типа <code>unsigned long int</code> (4294967295)
<code>LLONG_MAX</code>	Максимальное значение для объектов типа <code>long long int</code> (9223372036854775807)
<code>LLONG_MIN</code>	Минимальное значение для объектов типа <code>long long int</code> (-9223372036854775807)
<code>ULLONG_MAX</code>	Максимальное значение для объектов типа <code>unsigned long long int</code> (18446744073709551615)

stdbool.h

Этот заголовочный файл содержит определения для работы с булевыми переменными (тип `_Bool`).

Идентификатор	Значение
<code>bool</code>	Взаимозаменяемое имя для базового типа данных <code>_Bool</code>
<code>true</code>	Определено как 1
<code>false</code>	Определено как 0

float.h

Этот заголовочный файл содержит определения различных значений, связанных с вещественной арифметикой. Значения указываются в круглых скобках после описания. Не все определения, связанные с вещественной арифметикой, перечислены ниже.

Идентификатор	Значение
<code>FLT_DIG</code>	Количество цифр для типа <code>float</code> (6)
<code>FLT_EPSILON</code>	Наименьшее значение, которое не учитывается при суммировании с 1.0 ($1e-5$)
<code>FLT_MAX</code>	Максимальное значение для типа <code>float</code> ($1e+37$)
<code>FLT_MAX_EXP</code>	Максимальное значение для типа <code>float</code> ($1e+37$)
<code>FLT_MIN</code>	Минимальное значение для нормализованного типа <code>float</code> ($1e-37$)

Подобные определения существуют и для типов `double` и `long double`. Необходимо только заменить первые символы `FLT` на `DBL` для типа `double`, и `LDBL` для типа `long double`. Например, идентификатор `DBL_DIG` можно использовать для получения количества цифр, составляющих тип `double`, а идентификатор `LDBL_DIG` поможет определить количество цифр в типе `long double`.

Также необходимо отметить, что для получения дополнительной информации относительно вещественных типов и необходимых для работы с ними функций, лучше использовать заголовочный файл `fenv.h`. Например, в нем описывается функция с именем `fesetround`, которая позволяет задать направление округления вещественного значения с помощью опций `FE_TONEAREST`, `FE_UPWARD`, `FE_DOWNWARD` или `FE_TOWARDZERO`. Вы также можете убрать, сгенерировать или проверить исключения для вещественных типов, используя функции `feclearexcept`, `feraiseexcept` и `fetestexcept`.

stdint.h

Этот заголовочный файл содержит различные определения и константы, которые можно использовать при работе с целыми числами в программах, независимых от системы. Например, выражение `typedef int32_t` можно использовать для объявления целочисленных переменных со знаком размером 32 бита, не уточняя при этом размер используемых целочисленных типов для системы, на которой будет запускаться программа. Аналогично, тип `int_least32_t` можно использовать для объявления целочисленных переменных размером по крайней мере 32 бита. Другие определения

позволяют выбрать наиболее быстросействующие типы. Для более подробного знакомства с этими типами вы можете просмотреть этот файл или обратиться к документации.

Несколько полезных определений из этого заголовочного файла приведены ниже.

Идентификатор	Значение
<code>intptr_t</code>	Целочисленное значение для хранения указателей
<code>uintptr_t</code>	Беззнаковое целочисленное значение для хранения указателей
<code>intmax_t</code>	Наибольший целочисленный тип со знаком
<code>uintmax_t</code>	Наибольший целочисленный тип без знака

Функции для работы со строками

Следующие функции выполняют операции с символьными массивами. При описании этих функций используются символы `s`, `s1` и `s2`, которые представляют указатели на символьный массив, оканчивающийся нулем, символ `c` используется как тип `int`, а символ `n` представляет тип `size_t` (определен в файле `stddef.h`). Для подпрограмм `strnxxx`, символы `s1` и `s2` могут указывать на символьный массив, который не оканчивается нулем.

Для использования этих функций в программе, к ней необходимо подключить заголовочный файл `string.h`:

```
#include <string.h>

char *strcat (s1, s2)
```

Объединение символьных строк `s1` и `s2`. При этом строка `s2` помещается за строку `s1`, после чего ставится нулевой символ. Функция возвращает строку `s1`.

```
char *strchr (s, c)
```

Производит поиск первого вхождения символа `c` в строке `s`. Если символ найден, возвращается указатель на этот символ, в противном случае возвращается нулевой указатель.

```
int strcmp (s1, s2)
```

Сравнивает строки `s1` и `s2`. Возвращаемое значение будет меньше нуля, если строка `s1` меньше строки `s2`, будет равно нулю, если строка `s1` равна `s2`, и будет больше нуля, если строка `s1` больше строки `s2`.

```
char *strcoll (s1, s2)
```

Подобно функции `strcmp`, за исключением того, что указатели `s1` и `s2` ссылаются на строки в текущем фрагменте.

```
char *strcpy (s1, s2)
```

Копирует строку `s2` в `s1`, возвращается `s1`.

```
char *strerror (n)
```

Возвращает сообщение об ошибке, связанное с ошибкой номер `n`.

```
size_t strcspn (s1, s2)
```

Подсчитывает максимальное количество символов в строке s1, которые не являются символами строки s2.

```
size_t strlen (s)
```

Возвращает максимальное количество символов в строке s, исключая нулевой символ.

```
char *strncat (s1, s2, n)
```

Копирует строку s2 в конец строки s1, пока не будет достигнут нулевой символ или не будут скопированы n символов. Возвращается строка s1.

```
int strncmp (s1, s2, n)
```

Выполняет то же, что и функция strcmp, за исключением того, что сравниваются только n символов.

```
char *strncpy (s1, s2, n)
```

Копирует строку s2 в строку s1, пока не будет достигнут нулевой символ или не будут скопированы n символов. Возвращается строка s1.

```
char *strrchr (s, c)
```

Производит поиск последнего вхождения символа c в строке s. Если символ найден, возвращается указатель на этот символ, в противном случае возвращается нулевой указатель.

```
char *strpbrk (s1, s2)
```

Находит первое вхождение любого символа из строки s2 в строке s1. Возвращается указатель на этот символ, если поиск успешный, в противном случае возвращается нулевой указатель.

```
size_t strspn (s1, s2)
```

Подсчитывает максимальное количество символов в строке s1, которые одновременно являются символами строки s2. Возвращается количество символов.

```
char *strstr (s1, s2)
```

Находит первое вхождение строки s2 в строке s1. Возвращается указатель на начальный символ найденной строки, если поиск успешный, в противном случае возвращается нулевой указатель.

```
char *strtok (s1, s2)
```

Разбивает строку s1 на отдельные лексемы в соответствии с разделителями, указанными в строке s2. При первом вызове будет произведен синтаксический анализ строки s1. Функция помещает нулевые символы в строку s1 для того, чтобы отметить конец каждой лексемы, и возвращает указатели на начало лексем. При последовательных вызовах после выделения всех лексем возвращается нулевой указатель.

```
size_t strxfrm (s1, s2, n)
```

Трансформирует n символов строки s2, помещая результат в строку s1. Трансформированные строки могут сравниваться с помощью функции strcmp.

Работа с памятью

Следующие функции выполняют операции с символьными массивами. Они разработаны для эффективного поиска в памяти и для копирования одного пространства памяти в другое. Для использования этих функций в программе, к ней необходимо подключить заголовочный файл `string.h`.

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

При описании этих функций используются переменные `m1` и `m2` типа `void*`, переменная `c` типа `int`, которая может преобразовываться в тип `unsigned char`, и целочисленная переменная `n` типа `size_t`.

```
void *memchr (m1, c, n)
```

Поиск значения `c` в последовательности, на которую указывает переменная `m1`. Если значение найдено, возвращается указатель на это значение, а если после просмотра `n` символов ничего не найдено — нулевой указатель.

```
void *memcmp (m1, m2, n)
```

Сравнение соответствующих первых `n` символов из последовательностей, на которые ссылаются указатели `m1` и `m2`. Если символы идентичны, то возвращается нуль. Если нет, то возвращается значение, свидетельствующее о том, что символы не совпадают. Таким образом, если не совпадающий символ последовательности `m1` меньше, чем соответствующий символ последовательности `m2`, то возвращается значение, которое меньше нуля, в противном случае возвращается значение, которое больше нуля.

```
void *memcpy (m1, m2, n)
```

Копирует `n` символов из последовательности `m2` в последовательность `m1`. Возвращается `m1`.

```
void *memmove (m1, m2, n)
```

Подобно `memcpy`, но работает гораздо более гарантированно даже если последовательности `m1` и `m2` перекрываются в памяти.

```
void *memset (m1, c, n)
```

Присваивает первым `n` символам последовательности `m1` значение `c`. Возвращается указатель `m1`.

Обратите внимание, что функция не реагирует на нулевой символ внутри последовательности. Также она может использоваться с массивами, отличными от символьных, используя приведение к типу `void*`. Поэтому если указатели `data1` и `data2` ссылаются на массивы из 100 целых чисел, то вызов функции

```
memcpy ((void *) data2, (void *) data1, sizeof (data1));
```

приведет к тому, что будут скопированы все 100 значений из `data1` в `data2`.

Функции для работы с символами

Следующие функции предназначены для работы с одиночными символами. Для работы с ними необходимо подключить к программе файл `ctype.h`.

```
#include <ctype.h>
```

Каждая из перечисленных ниже функций принимает в качестве аргумента значение типа `int (c)` и возвращает значение `TRUE` (не ноль), если проверка завершается успешно, или значение `FALSE` (ноль) — в противном случае.

Имя	Проверка
<code>isalnum</code>	Символ <code>c</code> является буквенно-цифровым символом?
<code>isalpha</code>	Символ <code>c</code> является буквой?
<code>isblank</code>	Символ <code>c</code> является символом пробела (<code>space</code> или <code>tab</code>)?
<code>iscntrl</code>	Символ <code>c</code> является управляющим символом?
<code>isdigit</code>	Символ <code>c</code> является цифрой?
<code>isgraph</code>	Символ <code>c</code> является графическим символом (все печатаемые символы, исключая <code>space</code>)?
<code>islower</code>	Символ <code>c</code> является буквой в нижнем регистре?
<code>isprint</code>	Символ <code>c</code> является печатаемым символом (включая <code>space</code>)?
<code>ispunct</code>	Символ <code>c</code> является символом пунктуации (все символы, исключая <code>space</code> и буквенно-цифровые)?
<code>isspace</code>	Символ <code>c</code> является символом свободного места (<code>space</code> , новая строка, перевод каретки, горизонтальная и вертикальная табуляция, подача страницы)?
<code>isupper</code>	Символ <code>c</code> является буквой в верхнем регистре?
<code>isxdigit</code>	Символ <code>c</code> является шестнадцатеричной цифрой?

Следующие функции выполняют преобразование символов.

```
int tolower(c)
```

Возвращает эквивалентное значение для символа `c` в нижнем регистре. Если `c` уже находится в нижнем регистре, то возвращается само `c`.

```
int toupper(c)
```

Возвращает эквивалентное значение для символа `c` в верхнем регистре. Если `c` уже находится в верхнем регистре, то возвращается само `c`.

Функции ввода-вывода

В этом разделе описаны наиболее часто используемые функции ввода-вывода данных из библиотек языка C. Для использования этих функций в программе, к ней необходимо подключить заголовочный файл `stdio.h`.

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

В этот файл включены объявления для функций ввода-вывода и определения аббревиатур EOF, NULL, stdin, stdout, stderr (значения всех констант) и FILE.

В объявлениях используются указатели на строки к нулевым окончанием: fileName, fileName1, fileName2, accessMode и format. указатель buffer ссылается на символьный массив, указатель filePtr является указателем на тип "указатель на FILE.", переменные n и size являются положительными целочисленными значениями типа size_t, а переменные i и c имеют тип int.

```
void clearerr (filePtr)
```

Очищает конец файла и индикаторы ошибок, связанные с файлом, на который ссылается указатель filePtr.

```
int fclose (filePtr)
```

Закрывает файл, на который ссылается указатель filePtr, и возвращает нуль, если операция прошла успешно, в противном случае возвращается EOF.

```
int feof (filePtr)
```

Возвращает не нулевое значение, если счетчик считанных байтов указанного файла достиг конца файла, в противном случае возвращается нуль.

```
int ferror (filePtr)
```

Производит проверку на наличие ошибок в указанном файле и возвращает нуль, если ошибки есть, в противном случае возвращается не нулевое значение.

```
int fflush (filePtr)
```

Извлекает все данные из буфера, связанного с указанным файлом, и возвращает нуль, если операция прошла успешно, в противном случае возвращается EOF.

```
int fgetc (filePtr)
```

Возвращает очередной символ из файла, на который ссылается указатель filePtr, или значение EOF, если достигнут конец файла (обратите внимание, что эта функция возвращает тип int).

```
int fgetpos (filePtr, fpos)
```

Возвращает значение (позицию) указателя файла, на который ссылается указатель filePtr, сохраняя значение в переменной, на которую ссылается указатель fpos (определен в stdio.h). Возвращает нуль, если операция прошла успешно, в противном случае возвращается не нулевое значение. См. также функцию fsetpos.

```
char *fgets (buffer, i, filePtr)
```

Считывает символы из указанного файла до тех пор, пока не будет считано i-1 символов или пока не будет считан символ новой строки. Считанные символы сохраняются в символьном массиве, на который ссылается указатель buffer. Если считывается символ новой строки, он сохраняется в массиве. Если достигнут конец файла, то возникает ошибка и при этом возвращается значение NULL, в противном случае возвращается указатель buffer.

```
FILE *fopen (fileName, accessMode)
```

Открывает файл с указанным именем, используя заданный режим доступа. Разрешенными режимами являются: "r" для записи, "w" для чтения, "a" для добавления

данных в конец существующего файла, "r+" для записи/чтения с начала существующего файла, "w+" для записи/чтения, при этом предыдущее содержимое файла будет потеряно, и "a+" для записи/чтения, причем запись производится в конец файла. Если открываемый файл не существует, то он создается при режимах записи "w" "w+" и добавления "a" "a+". Если файл открывается в режиме добавления ("a" и "a+"), то предположительно существующие данные будут заменяться новыми данными.

Для систем, которые различают двоичные и текстовые файлы, символ b должен добавляться к режиму (например "rb"), с тем чтобы открыть файл в двоичном режиме.

Если вызов функции `fopen` прошел успешно, то указатель `FILE` возвращается и может использоваться для идентификации файла в операциях ввода/вывода, в противном случае возвращается нулевой указатель.

```
int fprintf (filePtr, format, arg1, arg2, ..., argn)
```

Записывает указанные аргументы в файл, на который ссылается указатель `filePtr`, в соответствии с форматом, заданным символьной строкой `format`. Форматирование символов производится так же, как и для функции `printf` (см. главу 16, "Операции ввода-вывода"). Возвращается количество записанных символов. Если возвращается отрицательное значение, то это говорит об ошибке.

```
int fputc (c, filePtr)
```

Записывает значение переменной `c` (приводится к типу `unsigned char`) в файл, на который ссылается указатель `filePtr`. Возвращается значение `c`, если запись прошла успешно, в противном случае возвращается EOF.

```
int fputs (buffer, filePtr)
```

Записывает символы из массива, на который ссылается указатель `buffer`, в заданный файл до тех пор, пока не будет достигнут нулевой символ в массиве. Символ новой строки в файл не записывается. В случае ошибки записи возвращается значение EOF.

```
size_t fread (buffer, size, n, filePtr)
```

Считывает `n` элементов данных из указанного файла в `buffer`. Каждый элемент данных имеет размер `size` в байтах. Например вызов.

```
numread = fread (text, sizeof (char), 80, in_file);
```

считает 80 символов из файла, который идентифицируется переменной `in_file`, и сохраняет их в массиве, на который ссылается указатель `text`. Функция возвращает число успешно считанных элементов данных.

```
FILE *freopen (fileName, accessMode, filePtr)
```

Закрывает файл, связанный с указателем `filePtr`, и открывает файл с именем `fileName` в заданном режиме доступа (см. функцию `fopen`). Открытый файл затем связывается с указателем `filePtr`. Если вызов функции `freopen` прошел успешно, то возвращается указатель `filePtr`, в противном случае возвращается нулевой указатель. Функция `freopen` часто используется для переназначения файлов `stdin`, `stdout` или `stderr`. Например, вызов.

```
if ( freopen ("inputData", "r", stdin) == NULL ) {  
    ...  
}
```


будет иметь эффект перенаправления потока `stdin` в файл `inputData`, который открывается в режиме чтения. Последующие операции ввода-вывода для потока `stdin` будут выполняться файлом `inputData` аналогично тому, как файл перенаправляется при вызове программы.

```
int fscanf (filePtr, format, arg1, arg2, ..., argn)
```

Считывает элементы данных из файла, на который ссылается указатель `filePtr`, в соответствии с форматом, который задается символьной строкой `format`. Считанные значения сохраняются в аргументах, заданных после строки форматирования, каждый из которых является указателем. Разрешенные для строки форматирования символы совпадают с теми, что разрешены для функции `scanf` (см. главу 16). Функция `fscanf` возвращает количество успешно считанных и присвоенных элементов (исключая присваивания `%n`), или значение EOF, если конец файла был достигнут до того, как был присвоен первый элемент.

```
int fseek (filePtr, offset, mode)
```

Устанавливает счетчик байтов (позиция) в указанном файле в соответствии со значением, задаваемым аргументом `offset` (тип `long int`). Отсчет ведется от начала или конца файла, в зависимости от значения аргумента `mode` (целое число). Если это значение равно `SEEK_SET`, отсчет ведется от начала файла. Если значение равно `SEEK_CUR`, отсчет ведется от текущего положения счетчика конца файла. Если значение равно `SEEK_END`, отсчет ведется от конца файла. Значения для идентификаторов `SEEK_SET`, `SEEK_CUR` и `SEEK_END` определены в файле `stdio.h`.

В системах, которые различают текстовые и бинарные файлы, значение `SEEK_END` может не поддерживаться для бинарных файлов. Для текстовых файлов `offset` должен быть равен или нулю, или значению, возвращаемому при предыдущем вызове функции `ftell`. В последнем случае переменная `mode` должна иметь значение `SEEK_SET`.

При успешном вызове функции `fseek` возвращается не нулевое значение.

```
int fsetpos (filePtr, fpos)
```

Устанавливает текущую позицию файла (счетчик), на который ссылается указатель `filePtr`, присваивая значение, на которое ссылается указатель `fpos`, который имеет тип `fpos_t` (определен в `stdio.h`). При успешном вызове функции возвращается нулевое значение, а в случае ошибки. — не нулевое. См. также функцию `fgetpos`.

```
long ftell (filePtr)
```

Возвращает относительное смещение в байтах текущей позиции файла, который идентифицируется указателем `filePtr`, или возвращает значение `-1L` в случае ошибки.

```
size_t fwrite (buffer, size, n, filePtr)
```

Записывает `n` элементов данных из буфера `buffer` в указанный файл. Каждый элемент данных имеет размер `size` в байтах. Возвращает количество успешно записанных элементов.

```
int getc (filePtr)
```

Считывает и возвращает очередной символ из указанного файла. Значение EOF возвращается в том случае, если достигнут конец файла или произошла ошибка.

```
int getchar (void)
```

Считывает и возвращает очередной символ из стандартного файла ввода `stdin`. Значение EOF возвращается в том случае, если достигнут конец файла или произошла ошибка.

```
char *gets (buffer)
```

Считывает `n` символов из стандартного файла ввода `stdin` в буфер `buffer` до тех пор, пока не будет достигнут символ новой строки. Символ новой строки в буфере не сохраняется, а символьная строка завершается нулевым символом. Если при выполнении чтения происходит ошибка или не считано никакого символа, то возвращается нулевой указатель, в противном случае возвращается `buffer`.

```
void perror (message)
```

Выводит в поток `stderr` объяснение последней ошибки, на которую указывает переменная `message`. Например, код

```
#include <stdlib.h>
#include <stdio.h>
if ( (in = fopen ("data", "r")) == NULL ) {
    perror ("data file read");
    exit (EXIT_FAILURE);
}
```

выводит сообщение об ошибке, если ошибка произошла при открытии файла `fopen`, предоставляя более подробную информацию пользователю о причине ошибки.

```
int printf (format, arg1, arg2, ..., argn)
```

Выводит указанные аргументы в поток `stdout` в соответствии с форматом, определяемым символьной строкой `format` (см. главу 16). Возвращает количество записанных символов.

```
int putc (c, filePtr)
```

Записывает значение переменной `c` типа `unsigned char` в указанный файл. В случае успешной записи возвращается значение `c`, в противном случае возвращается значение EOF.

```
int putchar(c)
```

Записывает значение переменной `c` типа `unsigned char` в стандартный выходной поток `stdout`. В случае успешной записи возвращается значение `c`, в противном случае возвращается значение EOF.

```
int puts (buffer)
```

Записывает символы, сохраняемые в буфере `buffer`, в поток `stdout` до тех пор, пока не встретится нулевой символ. Символ новой строки записывается как последний символ (в отличие от функции `fputs`). В случае ошибки возвращается значение EOF.

```
int remove (fileName)
```

Удаляет указанный файл. В случае ошибки возвращается не нулевое значение.

```
int rename (fileName1, fileName2)
```

Переименовывает файл с именем `fileName1` в имя `fileName2`. В случае ошибки возвращается не нулевое значение.

```
void rewind (filePtr)
```

Переустанавливает указанный файл в начало.

```
int scanf (format, arg1, arg2, ..., argn)
```

Считывает элементы из стандартного входного потока `stdin` в соответствии с форматом, задаваемым строкой `format` (см. главу 16). Все аргументы, следующие за строкой `format`, должны быть указателями. Количество элементов, которые были успешно считаны и присвоены (исключая присваивание `%n`), возвращается функцией. Возвращается значение EOF при достижении конца файла еще до того, как элемент будет преобразован.

```
FILE *tmpfile (void)
```

Создает и открывает временный бинарный файл для записи в режиме обновления (режим `"r+b"`). При возникновении ошибки возвращается значение `NULL`. Временный файл автоматически удаляется при завершении программы. Функция с именем `tmpnam` также способна создавать уникальные имена временных файлов.

```
int ungetc (c, filePtr)
```

Эффективно "помещает обратно" символ `c` в указанный файл. Символ не записывается обратно в файл, а только помещается в буфер, связанный с данным файлом. Следующий вызов функции `getc` возвращает именно этот символ. Функция `ungetc` может вызываться для "помещения обратно" только одного символа. Это значит, что должна быть произведена операция считывания, прежде чем вызывать функцию `ungetc` еще раз. Функция возвращает значение символа `c`, если символ был успешно помещен обратно, в противном случае возвращается значение EOF.

Функции преобразования форматов в памяти

Функции `sprintf` и `sscanf` используются для проведения преобразований форматов в памяти. Эти функции аналогичны функциям `fprintf` и `fscanf`, за тем исключением, что вместо первого аргумента, указывающего на файл `FILE`, ставится символьная строка. Для того чтобы использовать эти функции в программе, необходимо подключить заголовочный файл `stdio.h`.

```
int sprintf (buffer, format, arg1, arg2, ..., argn)
```

Заданные аргументы преобразовываются в соответствии с форматом, указанным в символьной строке `format` (см. главу 16) и помещаются в символьный массив, на который ссылается указатель `buffer`. Нулевой символ автоматически помещается в конец строки в массиве `buffer`. Возвращается количество символов, помещенных в массив, исключая завершающий нулевой символ. Например, в следующих строках

```
int version = 2;
char fname[125];
...
sprintf (fname, "/usr/data%i/2005", version);
```

результат обработки символьной строки `"/usr/data2/2005"` будет помещен в буфер `fname`.

```
int sscanf (buffer, format, arg1, arg2, ..., argn)
```

Значения считываются из буфера `buffer` в соответствии со строкой форматирования `format`, и размещаются в аргументах, следующих за `format` (см. главу 16). Количество успешно преобразованных элементов возвращается функцией. Например, с помощью следующих утверждений

```
char buffer[] = "July 16, 2004", month[10];
int day, year;
...
sscanf (buffer, "%s %d, %d", month, &day, &year);
```

строка "July" будет сохранена в переменной `month`, целочисленное значение 16 будет сохранено в переменной `day`, а целочисленное значение 2004 будет сохранено в переменной `year`. Следующий код

```
#include <stdio.h>
#include <stdlib.h>
if ( sscanf (argv[1], "%f", &fval) != 1 ) {
    fprintf (stderr, "Bad number: %s\n", argv[1]);
    exit (EXIT_FAILURE);
}
```

преобразует первый аргумент в командной строке (на который ссылается `argv[1]`) в указатель на вещественное число и проверяет значение, возвращаемое функцией `sscanf` с тем, чтобы удостовериться в успешном считывании аргумента. В следующих разделах описанные функции, которые также преобразуют строки в числа.

Преобразование строка-значение

Следующие функции преобразовывают символьные строки в числа. Для использования этих функций в программе, необходимо подключить заголовочный файл `stdio.h`.

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

В следующих описаниях указатель `s` ссылается на строку с нулевым окончанием, указатель `end` ссылается на указатель на символ, и переменная `base` имеет тип `int`.

Все функции пропускают пробелы и символы табуляции в начале строки и прекращают сканировать строку при достижении символа, который не может быть преобразован в указанный тип.

```
double atof (s)
```

Преобразовывает строку, на которую ссылается указатель `s`, в вещественное число, возвращая результат.

```
int atoi (s)
```

Преобразовывает строку, на которую ссылается указатель `s`, в целое число, возвращая результат.

```
int atol (s)
```

Преобразовывает строку, на которую ссылается указатель `s`, в число типа `long int`, возвращая результат.

```
int atoll (s)
```

Преобразовывает строку, на которую ссылается указатель *s*, в число типа `long int`, возвращая результат.

```
double strtod (s, end)
```

Преобразовывает строку, на которую ссылается указатель *s*, в тип `double` число, возвращая результат. Указатель на символ, который прекращает сканирование строки, выделяется с помощью указателя *end*, который не должен быть нулевым указателем.

Для примера рассмотрим следующий код.

```
#include <stdlib.h>
...
char buffer[] = " 123.456xyz", *end;
double value;
...
value = strtod (buffer, &end);
```

В этом случае должно произойти сохранение значения 123.456 в переменной *value*. Указатель на символ *end* используется в функции `strtod` для того, чтобы ссылаться на символ в буфере, при считывании которого должно прекратиться сканирование. В данном случае это будет символ "x".

```
float strtof (s, end)
```

Аналогична функции `strtod`, за исключением того, что преобразование будет производиться в тип `float`.

```
long int strtol (s, end, base)
```

Преобразовывает строку *s* в тип `long int`, возвращая результат преобразования. Здесь переменная *base* представляет целое значение между 2 и 36 включительно. Это целое число интерпретируется как базовое число. Если переменная *base* равна 0, результат может быть десятичным, восьмеричным (с предшествующим 0) или шестнадцатеричным числом, (предшествующие 0x или 0X). Если *base* равно 16, результирующему значению может предшествовать 0x или 0X.

Указатель на символ, который прекращает сканирование строки, выделяется с помощью указателя *end*, который должен быть не нулевым.

```
long double strtold (s, end)
```

Аналогична функции `strtod`, за исключением того, что преобразование будет производиться в тип `long double`.

```
long long int strtoll (s, end, base)
```

Аналогично функции `strtol`, за исключением того, что преобразование будет производиться в тип `long long double`.

```
unsigned long int strtoul (s, end, base)
```

Преобразовывает строку *s* в тип `unsigned long int`, возвращая результат преобразования. Остальные аргументы трактуются так, как в функции `strtol`.

```
unsigned long long int strtoull (s, end, base)
```

Преобразовывает строку *s* в тип `unsigned long long int`, возвращая результат преобразования. Остальные аргументы трактуются так, как в функции `strtol`.

Функции динамического размещения памяти

Следующие функции используются для динамического размещения и освобождения памяти. При описании этих функций переменные `n` и `size` представляют целочисленные значения типа `size_t`, а указатель `pointer` ссылается на тип `void`. При использовании этих функций в вашей программе, к ней необходимо дописать следующую строку.

```
#include <stdlib.h>

void *calloc (n, size)
```

Выделяет непрерывное пространство для `n` элементов данных, где каждый из элементов имеет длину `size` в байтах. Все элементы инициализируются нулями. В случае успешного выделения возвращается указатель на выделенное пространство, в противном случае возвращается нулевой указатель.

```
void free (pointer)
```

Возвращает блок памяти, на который ссылается указатель `pointer`, который был предварительно получен при вызове функций `calloc`, `malloc` или `realloc`.

```
void *malloc (size)
```

Выделяет непрерывное пространство размером `size` байтов. В случае успешного выделения возвращается указатель на выделенное пространство, в противном случае возвращается нулевой указатель.

```
void *realloc (pointer, size)
```

Изменяет размер предварительно размещенного в памяти блока в соответствии со значением `size` в байтах, возвращая указатель на новый блок, который может быть перемещен в другое место памяти, если нельзя создать непрерывный участок на основе старого блока, или нулевой указатель в случае ошибки.

Математические функции

В приведенном ниже списке перечислены математические функции. Для использования этих функций в программе необходимо подключить к программе файл `math.h`.

```
#include <math.h>
```

В стандартном заголовочном файле `tgmath.h` определены макроопределения для типов, которые можно использовать для вызова функций из библиотек для математических функций или библиотек функций для работы с комплексными числами. При этом не надо будет думать о точном соответствии типов. Например, можно использовать шесть различных функций для вычисления квадратного корня для различных типов аргументов и возвращаемых типов.

- `double sqrt (double x)`
- `float sqrtf (float x)`
- `long double sqrtl (long double x)`

- `double complex csqrt (double complex x)`
- `float complex csqrtf (float complex f)`
- `long double complex csqrtl (long double complex)`

Вместо того, чтобы разбираться, какая из шести функций подойдет в отдельном случае, вы просто подключаете файл `tgmath.h` вместо файла `math.h` или `complex.h` и используете обобщающую (generic) версию для этих функций с именем `sqrt`. Соответствующее макроопределение из файла `tgmath.h` подберет корректную функцию для этого случая.

Возвращаясь к файлу `math.h`, обратите внимание на то, что следующее макроопределение может использоваться для проверки специфических свойств вещественных значений, представленных аргументом.

```
int fpclassify (x)
```

При этом `x` может классифицироваться как не число (NaN) (`FP_NAN`), бесконечное (`FP_INFINITE`), нормализованное (`FP_NORMAL`), или субнормализованное число (`FP_SUBNORMAL`), нуль (`FP_ZERO`), или некоторыми другими категориями `FP_...`, которые определены в файле `math.h` и зависят от реализации.

Математические библиотеки содержат версии для типов `float`, `double` и `long double`, которые возвращают значения типов `float`, `double` и `long double`. Далее описаны версии для типов `double`. Версия для типов `float` имеет то же самое имя, но с буквой `f` в конце (например, `acosf`), а версия для типов `long double` имеет в конце букву `l` (например, `acosl`).

```
int isfin (x)
```

Переменная `x` представляет конечное значение?

```
int isinf (x)
```

Переменная `x` представляет бесконечное значение?

```
int isgreater (x, y)
```

Выполняется неравенство `x > y`?

```
int isgreaterequal (x, y)
```

Выполняется неравенство `x >= y`?

```
int islessequal (x, y)
```

Выполняется неравенство `x <= y`?

```
int islessgreater (x, y)
```

Выполняется неравенство `x < y` или `x > y`?

```
int isnan (x)
```

Переменная `x` не является числом (NaN)?

```
int isnormal (x)
```

Переменная `x` представляет нормализованное значение?

```
int isunordered (x, y)
```

Переменные x и y не упорядочены (например, одна или обе могут быть NaN)?

```
int signbit (x)
```

Знак переменной x отрицательный?

В следующих функциях переменные x , y и z имеют тип `double`, переменная r выражает значение угла в радианах и имеет тип `double`, а переменная n имеет тип `int`. Для получения более подробной информации об ошибках, которые могут возникнуть при выполнении этих функций, обращайтесь к документации.

```
double acos (x)
```

Возвращает арккосинус для переменной x , для которой соответствующий угол будет выражен в радианах в диапазоне $[0, \pi]$. Переменная x задается в диапазоне $[-1, 1]$.

```
double acosh (x)
```

Возвращает гиперболический арккосинус для переменной x , где $x \geq 1$.

```
double asin (x)
```

Возвращает арксинус для переменной x , для которой соответствующий угол будет выражен в радианах в диапазоне $[-\pi/2, \pi/2]$. Переменная x задается в диапазоне $[-1, 1]$.

```
double asinh (x)
```

Возвращает гиперболический арксинус для переменной x .

```
double atan (x)
```

Возвращает арктангенс для переменной x , для которой соответствующий угол будет выражен в радианах в диапазоне $[-\pi/2, \pi/2]$.

```
double atanh (x)
```

Возвращает гиперболический арктангенс для переменной x , где переменная $|x| \leq 1$.

```
double atan2 (y, x)
```

Возвращает арктангенс отношения y/x , для которого соответствующий угол будет выражен в радианах в диапазоне $[-\pi, \pi]$.

```
double ceil (x)
```

Возвращает наименьшее целочисленное значение, большее или равное x . Обратите внимание, что возвращаемое значение имеет тип `double`.

```
double copysign (x, y)
```

Возвращает значение, которое равно x и знак которого такой же, как у y .

```
double cos (r)
```

Возвращает косинус для переменной r .

```
double cosh (x)
```

Возвращает гиперболический косинус для переменной r .

```
double erf (x)
```


Рассчитывает и возвращает функцию ошибки для x .

`double erfc (x)`

Рассчитывает и возвращает дополнительную функцию ошибки для x .

`double exp (x)`

Возвращает e^x .

`double expm1 (x)`

Возвращает $e^x - 1$.

`double fabs (x)`

Возвращает абсолютное значение переменной x .

`double fdim (x, y)`

Возвращает $x - y$, если $x > y$, в противном случае возвращается 0.

`double floor (x)`

Возвращает наибольшее целочисленное значение, меньшее или равное x . Обратите внимание, что возвращаемое значение имеет тип `double`.

`double fma (x, y, z)`

Возвращает $(x * y) + z$.

`double fmax (x, y)`

Возвращает максимальное значение из x и y .

`double fmin (x, y)`

Возвращает минимальное значение из x и y .

`double fmod (x, y)`

Возвращает вещественный остаток от деления x на y . Знак результата такой же, как у x .

`double frexp (x, exp)`

Делит x на нормализованное значение и степень числа 2. Возвращает дробное значение в диапазоне $[1/2, 1]$ и сохраняет экспоненту в целочисленном значении, на которое указывает переменная `exp`. Если x равно нулю, возвращаются нормализованное значение и экспонента как нуль.

`int hypot (x, y)`

Возвращает квадратный корень суммы $x^2 + y^2$.

`int ilogb (x)`

Извлекает экспоненту из значения x как целочисленное значение со знаком.

`double ldexp (x, n)`

Возвращает $x * 2^n$.

`double lgamma (x)`

Возвращает натуральный логарифм абсолютного значения `gamma x`.

`double log (x)`

Возвращает натуральный логарифм `x`, $x \geq 0$.

`double logb (x)`

Возвращает экспоненту числа `x` со знаком.

`double loglp (x)`

Возвращает the натуральный логарифм значения `(x+1)`, $x \geq -1$.

`double log2 (x)`

Возвращает $\log_2 x$, $x \geq 0$.

`double log10 (x)`

Возвращает $\log_{10} x$, $x \geq 0$.

`long int lrint (x)`

Возвращает значение `x`, округленное до ближайшего целочисленного значения типа `long`.

`long long int llrint (x)`

Возвращает значение `x`, округленное до ближайшего целочисленного значения типа `long long`.

`long long int llround (x)`

Возвращает значение `x`, округленное до ближайшего значения типа `long long int`. Половинное значение всегда округляется от нуля (так 0.5 всегда округляется до 1).

`long int lround (x)`

Возвращает значение `x`, округленное до ближайшего значения типа `long int`. Половинное значение всегда округляется от нуля (так 0.5 всегда округляется до 1).

`double modf (x, ipart)`

Извлекает целую и дробную части переменной `x`. Возвращается дробная часть, а целая часть сохраняется в переменной, на которую ссылается указатель `ipart` типа `double`.

`double nan (s)`

Возвращает NaN, при этом на исходное значение ссылается указатель на строку `s`.

`double nearbyint (x)`

Возвращает значение `x`, округленное до ближайшего вещественного значения.

`double nextafter (x, y)`

Возвращает следующее репрезентативное значение `x` в направлении `y`.

`double nexttoward (x, ly)`

Возвращает следующее репрезентативное значение x в направлении y . Аналогична `nextafter`, за исключением того, что второй аргумент имеет тип `long double`.

```
double pow (x, y),
```

Возвращает x^y . Если x меньше нуля, y должен быть целым числом. Если x равен нулю, y должен быть больше нуля.

```
double remainder (x, y)
```

Возвращает остаток от деления x на y .

```
double remquo (x, y, quo)
```

Возвращает остаток от деления x на y , сохраняя частное в том месте, на которое ссылается указатель `quo`.

```
double rint (x)
```

Возвращает значение x , округленное до ближайшего целочисленного значения в вещественном формате. Может генерировать исключительную ситуацию, если результирующее значение не равно аргументу x .

```
double round (x)
```

Возвращает значение x , округленное до ближайшего целочисленного значения в вещественном формате. Половинное значение всегда округляется от нуля (так 0.5 всегда округляется до 1).

```
double scalbln (x, n)
```

Возвращает $x * FLT\_RADIX^n$, где n имеет тип `long int`.

```
double scalbn (x, n)
```

Возвращает $x * FLT\_RADIX^n$.

```
double sin (r)
```

Возвращает синус r .

```
double sinh (x)
```

Возвращает гиперболический синус x .

```
double sqrt (x)
```

Возвращает квадратный корень x , $x \geq 0$.

```
double tan (r)
```

Возвращает тангенс r .

```
double tanh (x)
```

Возвращает гиперболический тангенс x .

```
double tgamma (x)
```

Возвращает $\Gamma(x)$.

```
double trunc (x)
```

Усекает аргумент x до целого числа, возвращая результат как тип `double`.

Арифметика комплексных чисел

В заголовочном файле `complex.h` заданы различные типы определений и функций для работы с комплексными числами. Ниже перечислены несколько макроопределений из этого файла и приведен список функций для работы с комплексными числами.

Имя	Проверка
<code>complex</code>	Дополнительное имя для типа <code>_Complex</code>
<code>_Complex_I</code>	Макроопределение для задания комплексной части числа (например, <code>4+6.2*_Complex_I</code> задается как <code>4+6.2i</code>)
<code>imaginary</code>	Дополнительное имя для типа <code>_Imaginary</code> . Определяется в том случае, если реализация поддерживает комплексные числа
<code>_Imaginary_I</code>	Макроопределение для задания мнимой части числа.

В списке перечисленных ниже функций переменные `y` и `z` имеют тип `double complex`, переменная `x` имеет тип `double`, и переменная `n` задает тип `int`.

Математические библиотеки для комплексных чисел содержат версии для типов `float complex`, `double complex` и `long double complex`, которые возвращают значения типов `float complex`, `double complex` и `long double complex`. Далее описаны версии для типов `double complex`. Версия для типов `float complex` имеет то же самое имя, но с буквой `f` в конце (например, `cacosf`), а версия для типов `long double complex` имеет в конце букву `l` (например, `cacosl`).

```
double complex cabs (z)2
```

Возвращает комплексное абсолютное значение переменной `z`.

```
double complex cacos (z)
```

Возвращает комплексный арккосинус переменной `z`.

```
double complex cacosh (z)
```

Возвращает комплексный гиперболический арккосинус переменной `z`.

```
double carg (z)
```

Возвращает фазовый угол `z`.

```
double complex casin (z)
```

Возвращает комплексный арксинус переменной `z`.

```
double complex casinh (z)
```

Возвращает комплексный гиперболический арксинус переменной `z`.

```
double complex catan (z)
```

Возвращает комплексный арктангенс `z`.

```
double complex catanh (z)
```

Возвращает комплексный гиперболический арктангенс `z`.

```
double complex ccos (z)
```

Возвращает комплексный косинус z .

`double complex ccosh (z)`

Возвращает комплексный гиперболический косинус z .

`double complex cexp (z)`

Возвращает комплексную натуральную экспоненту z .

`double cimag (z)`

Возвращает мнимую часть z .

`double complex clog (z)`

Возвращает комплексный натуральный логарифм z .

`double complex conj (z)`

Возвращает комплексное сопряженное значение для z (инвертирует знак мнимой части).

`double complex cpow (y, z)`

Возвращает комплексную степень y (y^z).

`double complex cproj (z)`

Возвращает проекцию z на сферу Римана.

`double complex creal (z)`

Возвращает реальную часть z .

`double complex csin (z)`

Возвращает комплексный синус z .

`double complex csinh (z)`

Возвращает комплексный гиперболический синус z .

`double complex csqrt (z)`

Возвращает комплексный квадратный корень z .

`double complex ctan (z)`

Возвращает комплексный тангенс z .

`double complex ctanh (z)`

Возвращает комплексный гиперболический тангенс z .

Функции общего назначения

Это несколько функций из библиотеки, которые не входят в предыдущие категории. Для использования этих функций подключите файл `stdlib`.

```
int abs (n)
```

Возвращает абсолютное значение аргумента `n` типа `int`.

```
void exit (n)
```

Прекращает выполнение программы, закрывая все открытые файлы и возвращая состояние, выраженное целым числом. Значения `EXIT_SUCCESS` и `EXIT_FAILURE`, определенные в файле `stdlib.h`, могут использоваться для возврата успешного завершения или завершения с ошибкой.

Другие функции из библиотеки, которые можно использовать для подобных целей, это функции `abort` и `atexit`.

```
char *getenv (s)
```

Возвращает указатель на значение для переменной окружения, на которую ссылается указатель `s`, или возвращает нулевой указатель, если переменная не существует. Работа этой функции зависит от системы. Например, что касается системы Unix, следующий код

```
char *homedir;  
...  
homedir = getenv ("HOME");
```

может использоваться для получения значения пользовательской переменной `HOME`, сохраняя указатель на него в переменной `homedir`.

```
long int labs (l)
```

Возвращает абсолютное значение аргумента `l` типа `long int`.

```
long long int llabs (ll)
```

Возвращает абсолютное значение аргумента `ll` типа `long long int`.

```
void qsort (arr, n, size, comp_fn)
```

Сортирует массив данных, на который ссылается указатель `arr`. Здесь переменная `n` задает количество элементов массива, каждый из которых имеет размер `size` байтов. Переменные `n` и `size` имеют тип `size_t`. Четвертый аргумент имеет тип "указатель на функцию, которая возвращает тип `int` и принимает в качестве аргументов два указателя типа `void`". Функция `qsort` вызывает эту функцию для сравнения двух элементов массива, передавая указатели на элементы массива для сравнения в качестве аргументов. Эта функция обычно разрабатывается пользователем и после сравнения элементов должна возвращать значение которое меньше нуля, больше нуля или равно нулю, если первый элемент меньше второго, больше второго или равен второму соответственно.

Ниже показан пример использования функции `qsort` для массива из 1000 целых чисел с именем `data`.

```
#include <stdlib.h>
...
int main (void)
{
    int data[1000], comp_ints (void *, void *);
    ...
    qsort (data, 1000, sizeof(int), comp_ints);
    ...
}

int comp_ints (void *p1, void *p2)
{
    int i1 = * (int *) p1;
    int i2 = * (int *) p2;
    return i1 - i2;
}
```

Еще одна функция `bsearch`, которая не описана здесь, принимает аргументы аналогично функции `qsort` и выполняет бинарный поиск в упорядоченном массиве данных.

```
int rand (void)
```

Возвращает случайное число в диапазоне `[0, RAND_MAX]`, где `RAND_MAX` определен в файле `stdlib.h` и имеет минимальное значение 32767. См. также функцию `srand`.

```
void srand (seed)
```

Задаёт начальное значение для генератора случайных чисел, используя значение `seed` типа `unsigned int`.

```
int system (s)
```

Передаёт в систему команду для выполнения, содержащуюся в символьном массиве, на который ссылается указатель `s`. Если `s` будет нулевым указателем, то функция `system` возвратит не нулевое значение при том условии, что командный процессор сможет выполнить эту команду. Например, в системе Unix вызов

```
system ("mkdir /usr/tmp/data");
```

заставит систему создать каталог с именем `/usr/tmp/data` (предполагается, что у вас есть соответствующий допуск).

В

Компиляция программ с помощью gcc

В этом разделе приводятся некоторые наиболее часто используемые опции для работы с компилятором gcc. Для получения более подробной информации обо всех опциях командной строки при работе в операционной системе Unix, наберите команду `man gcc`. Также можете посетить сайт, посвященный gcc, по адресу: <http://gcc.gnu.org/onlinedocs>, где вы найдете полный комплект документации.

В этом приложении приведены опции командной строки для компилятора gcc версии 3.3, которые не включают расширения, добавленные другими разработчиками, такими как Apple Computer, Inc.

Формат команд

Общий формат для всех команд компилятора gcc следующий.

```
gcc [options] file [file ...]
```

Здесь элементы, заключенные в квадратные скобки, являются дополнительными.

Каждый файл, перечисленный в списке, обрабатывается компилятором. Это подразумевает препроцессорную обработку, компиляцию, компоновку и сборку. Опции командной строки могут использоваться в любой последовательности.

Способ, которым обрабатывается каждый входной файл, определяется по расширению файла. Это может быть переопределено в помощью опции `-x` (см. документацию). В табл. В.1 приведены основные расширения файлов.

Таблица В.1. Основные расширения файлов

Расширение	Значение
.c	Исходный файл на языке C
.cc, .cpp	Исходный файл на языке C++
.h	Заголовочный файл
.m	Исходный файл на языке Objective-C
.pl	Исходный файл на языке Perl
.o	Объектный файл (предварительно скомпилированный файл)

Опции командной строки

В табл. В.2 приведены опции, наиболее часто используемые с компилятором gcc.

Таблица В.2. Опции, наиболее часто используемые с gcc

Опция	Значение	Пример
--help	Отображает список опций командной строки	gcc --help
-c	Не компонует файлы, сохраняя для каждого исходного файла объектный файл с расширением .o	gcc -c enumerator.c
-dumpversion	Отображает текущую версию gcc	gcc -dumpversion
-g	Включает отладочную информацию для использования с отладчиком gdb (используйте -ggdb, если поддерживается несколько отладчиков)	gcc -g testprog.c -o testprog
-D id	Определяет идентификатор id для препроцессора со значением 1	
-D id=value	Определяет идентификатор id и устанавливает его значение	gcc -D DEBUG=3 test.c
-E	Только препроцессорная обработка и запись результатов в стандартный выходной поток. Используется для проверки работы препроцессора	gcc -E enumerator.c
-I dir	Добавляет каталог dir в список каталогов для поиска заголовочных файлов. Этот каталог будет просмотрен перед другими стандартными каталогами	gcc -I /users/ steve/include x.c
-ilibrary	Добавляет ссылку на файлы, используемые в библиотеках. Эта опция должна ставиться после файлов, которым необходимы библиотечные функции. Компоновщик ищет в заданных каталогах (см. опцию -L) файл с именем liblibrary.a	gcc mathfuncs. c -lm
-L dir	Добавляет каталог dir в список каталогов для поиска библиотечных файлов. Этот каталог будет просмотрен перед другими стандартными каталогами	gcc -L /users/ steve/lib x.c
-o execfile	Размещает исполняемый модуль в файле с именем, заменяющем execfile	gcc dbtest.c -o dbtest

Опция	Значение	Пример
-Olevel	Оптимизирует код в целях повышения скорости работы, задавая необходимый уровень level, который может принимать значения 1, 2 или 3. Если никакого уровня не задано, то по умолчанию будет выбрано значение 1. Большее значение задает лучшую оптимизацию, но при этом возрастет время компиляции и будут ограничены возможности отладки с помощью gdb	gcc -O3 m1.c m2.c -o mathfuncs
-std=standard	Задаёт стандарт для файлов на языке C. Используйте c99 для стандарта ANSI C99 без дополнений GNU	gcc -std=c99 mod1.c mod2.c
-Wwarning	Включает предупреждающие сообщения. Эта опция полезна всегда, т.к. предупреждающие сообщения позволяют избежать серьезных ошибок	gcc -Werror mod1. c mod2.c

Г

Часто встречающиеся ошибки

Ниже приведены наиболее часто встречающиеся в программах на языке C ошибки. Они никак не упорядочены. Знание этих ошибок поможет вам избежать многих неприятностей при написании ваших программ.

1. *Неправильно установлены точка с запятой.*

Пример

```
if ( j == 100 );  
    j = 0;
```

В предыдущих строках кода значение переменной `j` будет всегда устанавливаться в 0, чего не должно быть по логике рассуждения. Это происходит из-за того, что ошибочно поставлены точка с запятой после закрывающей скобки. В данном случае получается синтаксически правильная конструкция, т.к. точка с запятой после закрывающей скобки представляет нулевое утверждение, а значит, компилятор не будет воспринимать это как ошибку. Такой тип логической ошибки также часто встречается в циклах `while` и `for`.

2. *Ошибочное использование оператора `=` вместо оператора `==`.*

Такие ошибки обычно происходят в утверждениях `if`, `while` или `do`.

Пример

```
if ( a = 2 )  
    printf ("Your turn.\n");
```

Это утверждение полностью корректно и выполняет присваивание значения 2 переменной `a`, после чего вызывает функцию `printf`. При этом функция `printf` будет вызываться всегда, поскольку значение выражения условия для `if` будет всегда больше нуля (в данном случае оно всегда будет равно 2)

3. *Пропуск объявления прототипа.*

Пример

```
result = squareRoot (2);
```

Если функция `squareRoot` объявлена в программе позже этой строки или объявлена в другом файле, а явно не объявлена до этой строки, то компилятор предположит, что функция возвращает значение типа `int`. Более того, компилятор преобразует аргументы типа `float` в тип `double`, и аргументы типа `_Bool`, `char` и `short` в тип `int`.

Поэтому всегда лучше сделать прототипы объявлений для всех функций, которые вы используете (или явно опишите их, или подключите к программе заголовочный файл с корректными объявлениями), даже если они уже были объявлены.

4. Ошибка с учетом приоритетов различных операторов.

Пример

```
while ( c = getchar () != EOF )
    ...
if ( x & 0xF == y )
    ...
```

В первом случае значение, возвращаемое функцией `getchar`, сначала сравнивается со значением `EOF`, хотя вполне очевидно, что было задумано совсем другое. Это происходит потому, что оператор неравенства имеет более высокий приоритет, чем оператор присваивания. Поэтому значением, которое будет присвоено переменной `c`, будет результат сравнения, т.е. 1, если возвращаемое функцией `getchar` значение не равно `EOF`, и 0 — в противном случае.

Во втором случае целочисленная константа `0xF` сравнивается со значением `y`, поскольку оператор равенства имеет более высокий приоритет, чем все поразрядные операторы. Результатом сравнения будут значения 1 или 0, которые затем и будут использоваться в операции поразрядного И со значением `x`.

5. Ошибочное использование символьной константы и символьной строки.

В утверждении

```
text = 'a';
```

один символ присваивается переменной `text`. А в утверждении

```
text = "a";
```

переменной `text` присваивается указатель на строку "a". При этом в первом случае переменная `text` должна быть объявлена как тип `char`, а во втором случае она должна быть объявлена как тип "указатель на `char`".

6. Выход за границы массива.

Пример

```
int a[100], i, sum = 0;
...
for ( i = 1; i <= 100; ++i )
    sum += a[i];
```

Допустимые значения индекса должны находиться в диапазоне от 0 до числа элементов минус 1. А значит, в предыдущем цикле произойдет выход за пределы массива, поскольку последним значением индекса должно быть число 99, а не 100. Также обратите внимание, что для учета всех элементов массива необходимо начинать со значения индекса `i`, равного 0, а не 1.

7. Нет дополнительного места в символьном массиве для размещения заключительного нулевого символа.

Не забывайте объявлять размер символьного массива с учетом свободного места для размещения нулевого символа. Например, для символьной строки "hello" необходимо зарезервировать шесть мест, с учетом нулевого символа в конце.

8. Ошибочное использование оператора ">" вместо оператора "." при обращении к членам структуры.

Всегда помните, что оператор "." используется со структурными переменными, тогда как оператор ">" используется с указателями на структуры. Следовательно, если *x* является структурной переменной, то выражение *x.m* может использоваться для обращения к члену *m* структурной переменной *x*. С другой стороны, если *x* является указателем на структуру, то для обращения к члену *m* структуры, на которую ссылается *x*, необходимо написать: *x->m*.

9. Пропуск амперсанта перед переменной, не являющейся указателем, при вызове функции `scanf`.

Пример

```
int number;
...
scanf ("%i", number);
```

Помните, что все аргументы, которые располагаются в функции `scanf` после строки форматирования, должны быть указателями.

10. Использование неинициализированного указателя.

Пример

```
char *char_pointer;
*char_pointer = 'X';
```

Вы должны использовать указатель только после того, как ему будет присвоено некоторое значение, определяющее адрес переменной. В данном примере указатель `char_pointer` ни на что не указывает, и поэтому второе утверждение бессмысленно.

11. Пропуск ключевого слова `break` во фрагменте `case` для утверждения `switch`.

Помните, что если во фрагменте `case` нет ключевого слова `break`, то выполнение продолжится на следующий фрагмент `case`.

12. Размещение точки с запятой в конце директивы препроцессора.

Эту ошибку довольно часто допускают начинающие программисты из-за привычки ставить точку с запятой в конце каждого утверждения. Но в данном случае это приводит к тому, что все, что находится после имени в правой части директивы `#define`, непосредственно подставляется в программу. Поэтому описание

```
#define END_OF_DATA 999;
```

будет источником синтаксической ошибки, если это имя использовать в программе следующим образом.

```
if ( value == END_OF_DATA )
...
```

поскольку после препроцессора компилятор прочитает следующее утверждение.

```
if ( value == 999; )  
    ...
```

13. Пропуск скобок вокруг аргументов при описании макроопределения.

Пример

```
#define reciprocal(x) 1 / x  
...  
w = reciprocal ( a + b );
```

Предыдущее утверждение будет заменено некорректным выражением.

```
w = 1 / a + b;
```

14. Вставка пробела между именем макроопределения и его аргументом в директиве #define.

Пример

```
#define MIN (a,b) ( (a) < (b) ) ? (a) : (b) )
```

Это описание некорректно, т.к. препроцессор посчитает пробел после слова MIN как начало описания для имени MIN. В этом случае утверждение

```
minVal = MIN (val1, val2);
```

будет заменено следующим.

```
minVal = (a,b) ( (a) < (b) ) ? (a) : (b) ) (val1, val2);
```

А это совсем не то, что нам нужно.

15. Использование выражений, которые имеют двойное действие при вызове макроопределения.

Пример

```
#define SQUARE(x) (x) * (x)  
...  
w = SQUARE (++v);
```

При замене макроопределения SQUARE переменная *v* будет инкрементирована дважды, поскольку это утверждение будет расширено препроцессором следующим образом.

```
w = (++v) * (++v);
```

Д

Ресурсы

В этом приложении содержится список избранных ресурсов, который поможет вам в поиске необходимой информации. Необходимая информация может находиться как на Web-сайтах, так и в книгах. Если вы не можете найти нужные вам сведения, то обращайтесь ко мне по адресу: steve@kochan-wood.com, и я постараюсь вам помочь.

Ответы на вопросы, список опечаток и т.д.

Вы можете посетить Web-сайт www.kochan-wood.com, где представлены ответы на приведенные в книге упражнения, а также список опечаток. Также здесь можно найти последние новые справочные материалы.

Язык программирования C

Языку программирования C уже более 25 лет, и поэтому, конечно, нет недостатка в информации по этому предмету. Поэтому ниже представлены только некоторые источники этой информации.

Книги

Kernighan, Brian W., and Dennis M. Ritchie. *The C Programming Language*, 2nd Ed. Englewood-Cliffs, NJ: Prentice Hall, Inc., 1988.

Это была первая книга по языку C, написанная одним из разработчиков языка C.

Harbison, Samuel P. III, and Guy L. Steele Jr. *C: A Reference Manual*, 5th Ed. Englewood-Cliffs, NJ: Prentice Hall, Inc., 2002.

Еще один отличный справочник по языку C для программистов.

Plauger, P. J. *The Standard C Library*. Englewood-Cliffs, NJ: Prentice Hall, Inc., 1992.

В этой книге описаны стандартные библиотеки языка C, но, как можно видеть из даты публикации, в нее не включен стандарт ANSI C99. В частности, нет библиотек для работы с комплексными числами.

Web-сайты

www.kochan-wood.com

На этом Web-сайте вы найдете новое электронное издание книги *Topics in C Programming*, которую я написал совместно в Патриком Вудом в дополнение к моей оригинальной книге *Programming in C*.

www.ansi.org

Это официальный сайт организации ANSI. Здесь можно найти спецификации на языке C. Наберите 9899:1999 в окне поиска для получения адресов спецификации ANSI C99.

www.opengroup.org/onlinepubs/007904975/idx/index.html

Это большой электронный справочник по библиотечным функциям, где можно найти функции не только по стандарту ANSI.

Группы новостей

comp.lang.c

Эта группа новостей посвящена языку программирования C. Здесь можно задать вопрос и получить помощь других программистов. Также можно извлечь большую пользу, участвуя в дискуссиях. К этой группе новостей удобно получать доступ по адресу <http://groups.google.com>.

Компиляторы и интегрированные среды разработки

Ниже приводится список сайтов, где можно бесплатно скачать или купить компиляторы языка C и среды разработки, а также получить электронную документацию.

gcc

<http://gcc.gnu.org/>

Компилятор языка программирования C, разработанный организацией по созданию бесплатного программного обеспечения Free Software Foundation (FSF) и названный gcc. Этот компилятор использовался фирмой Apple для разработки операционной системы Mac OS X. С этого сайта можно загрузить бесплатную версию компилятора.

MinGW

www.mingw.org

Если вы только начинаете программировать на языке C в среде Windows, то вы можете получить компилятор gcc по лицензии GNU с этого сайта. Также можете бесплатно получить удобную для работы оболочку MSYS.

CygWin

www.cygwin.com

Фирма CygWin разрабатывает Linux-подобные средства для работы в среде Windows. Эти среды разработки распространяются на безоплатной основе.

Visual Studio

<http://msdn.microsoft.com/vstudio>

Visual Studio является интегрированной средой разработки от фирмы Microsoft, которая позволяет разрабатывать приложения на различных языках программирования.

CodeWarrior

www.metrowerks.com/mw/products/default.htm

Фирма Metrowerks предлагает профессиональные интегрированные среды разработки для различных операционных систем, включая Linux, Mac OS X, Solaris и Windows.

Kylix

www.borland.com/kylix/

Kylix является интегрированной средой разработки от фирмы Borland для разработки приложений в среде Linux.

Разное

В следующие разделы включены ресурсы для изучения объектно-ориентированного программирования и инструментальных средств разработчика.

Объектно-ориентированное программирование

Budd, Timothy. *The Introduction to Object-Oriented Programming, 3rd Ed.* Boston: Addison-Wesley Publishing Company, 2001.

Это классическая книга, в которой излагаются основы объектно-ориентированного программирования.

Язык C++

Prata, Stephen. *C++ Primer Plus, 4th Ed.* Indianapolis: Sams Publishing, 2001.

Учебное пособие, в котором описан язык программирования C++.

Stroustrup, Bjarne. *The C++ Programming Language, 3rd Ed.* Boston: Addison-Wesley Publishing Company, 2000.

Это классическая книга по языку программирования, написанная изобретателем языка C++.

Язык C#

Petzold, Charles. *Programming in the Key of C#.* Redmond, WA: Microsoft Press, 2003.

Это хорошая книга для тех, кто только начинает изучать язык программирования C#.

Liberty, Jesse. *Programming C#, 3rd Ed.* Cambridge, MA: O'Reilly & Associates, 2003.

Хорошее введение в язык программирования C# для опытных программистов.

Язык Objective-C

Kochan, Stephen. *Programming in Objective-C*. Indianapolis: Sams Publishing, 2004.

Эта книга представляет собой введение в язык Objective-C, не требуется предварительное знание языка C или принципов объектно-ориентированного программирования.

Apple Computer, Inc. *The Objective-C Programming Language*. Cupertino, CA: Apple Computer, Inc., 2004.

Это великолепный справочник по языку Objective-C для программистов, которые работают с языком C. Можно получить без оплаты в формате pdf по адресу:

<http://developer.apple.com/documentation/Cocoa/Conceptual/ObjectiveC/ObjC.pdf>.

Инструментарий разработчика

www.gnu.org/manual/manual.html

Здесь можно найти огромное количество полезных руководств, включая документацию по cvs, gdb, make и опциям командной строки для Unix.

Предметный указатель

А

Автоматическая локальная переменная 135, 163
Адрес 267
Аргумент 134, 197
Аргументы командной строки 367
Ассоциативность операторов 50

Б

Байт 267, 271
Бесконечный цикл 81
Бит 271
 младший 271
 наибольший значащий 271
 наименьший значащий 271
 старший 271
Битовое поле 284, 286
Блок 66, 440

В

Выравнивание по правому разряду 68
Выражение
 константное 43
 отношения 63

Д

Двойной связанный список 240
Двоичное дополнение 271
Двоичный поиск 220
Дерево 165, 240, 252
Динамическое выделение памяти 128, 246

З

Знаковое расширение 320

И

Имя формального параметра 135
Индекс 110
Инициализация символьных массивов 202
Инкапсуляция 402

К

Кавычки
 двойные 197
 одинарные 197
Класс памяти 436
Компиляция
 раздельная 324
 условная 307
Конец файла 349
Консоль 335
Константа 43
 вещественная 45
 символьная 45
Косвенная адресация 231

Л

Левостороннее выражение 423

М

Макроопределение 105, 300
Мантисса 44
Маска 285
Массив 109
 символьный 197

Метод 398
Минус
 унарный 54
Модификатор 48
 точности 84
Модульное программирование 323

Н

Наибольший общий делитель 74
Нулевой указатель 244

О

Область видимости 436
Объединение 363
Объект 397
Объявление прототипа функции 135
Операнд 49
Оператор
 if 81
 if-else 86
 адресации 232
 арифметический 49
 вывод в поток 406
 декремента 67
 деление по модулю 85
 инкремента 67
 отношения 63
 перегрузка 406
 поразрядная инверсия 277
 разыменования 232
 сдвиг влево 278
 сдвиг вправо 279
 тернарный 104
 условия 104

П

Переменная
 внешняя 326
 глобальная 160, 326
 локальная 72, 160
Плюс
 унарный 54
Показатель степени 44
Полиморфизм 399
Постдекремент 262
Постинкремент 262

Предекремент 262
Преинкремент 262
Префикс 45
Приведение типов 57
Приоритет 50
Программы
 самодокументируемые 42
Прототип объявления 439

Р

Рекурсия 165
Решето Арастофена 129

С

Свойство 408
Связанный список 240
Сдвиг вправо
 арифметический 279
 логический 279
Символ
 обратной черты 45
 переходной 214
 универсальный 216
 форматирования 43, 45, 46, 203, 206
Скобка
 круглая 33, 52
 фигурная 33, 66, 73
Слова
 зарезервированные 42
 ключевые 42
Составные литералы 184
Спецификатор 48
 ширины поля 68
Список 165
Стандартное арифметическое преобразование 435
Старшинство 50
Строка
 извлечение 198
 копирование 198
 нулевая 213
 объединение 198
 равенство 198
 символов 45
 символьная 193

Структура 171
 глобальная 178
 локальная 178

Т

Таблица истинности 273
Терминал 335
Тип
 возвращаемого значения 137
 перечислимый 313
Треугольное число 61
Триграф 444

У

Указатель 232
Утверждение
 составное 440

Ф

Файл
 заголовочный 304
 подключаемый 304
Факториал 165
Функция
 main 131

Х

Хеш-таблица 252

Ц

Целочисленное расширение 435
Функция
 calloc 128
 malloc 128

Л

Lvalue 423

Научно-популярное издание

Стефан Кочан

Программирование на языке С
3-е издание

Литературный редактор	<i>С.Г. Татаренко</i>
Верстка	<i>М.А. Смолина</i>
Художественный редактор	<i>С.А. Чернокозинский</i>
Корректор	<i>А.В. Луценко</i>

Издательский дом "Вильямс"
101509, г. Москва, ул. Лесная, д. 43, стр. 1

Подписано в печать 21.08.2006. Формат 70×100/16.
Гарнитура NewBaskerville. Печать офсетная.
Усл. печ. л. 39,99. Уч.-изд. л. 25,41.
Тираж 3000 экз. Заказ № 2615

Отпечатано по технологии StP
в ОАО "Печатный двор" им. А. М. Горького
197110, Санкт-Петербург, Чкаловский пр., 15.



Полное
введение в язык
программирования C

Программирование на языке C

Третье издание

Эта книга призвана научить читателя писать программы на языке C. Она окажет неоценимую услугу как для начинающих, так и для опытных программистов. В ней излагаются подробные сведения о языке программирования C, который является основой для многих языков программирования, таких как C++, Objective-C, C# и Java.

В книге приведены примеры завершенных программ, иллюстрирующих каждый новый материал. Автор шаг за шагом объясняет все возможности языка программирования C. Вы познакомитесь как с основами языка, так и с хорошей практикой написания программ. Упражнения в конце каждой главы делают книгу идеальным учебным пособием, которым могут пользоваться как учащиеся, так и преподаватели.

В этой книге раскрыты все возможности языка C, включая последние дополнения, сделанные в стандарте ANSI C99. В приложении кратко изложены все возможности языка и стандартные библиотеки, представленные в удобном для быстрого поиска информации виде.

Стефан Кочан является автором и соавтором шести классических книг по программированию и операционной системе UNIX. Он был ведущим консультантом по вопросам программирования в фирме AT&T Bell Laboratories, где разработал учебные курсы для специалистов по программированию на языке C и операционной системе UNIX.

*"Несомненно лучшая книга для тех, кто начинает
программировать на языке C. Ее отличают
последовательное изложение материала с удачными
примерами, и хороший технический язык. Эта книга
позволила мне понять язык C — поистине великая книга."*

Винит Карпентер, программист на C/C++

Языки программирования scan by @.Д.[].

ISBN 5-8459-1088-9



www.williamspublishing.com

DEVELOPER'S
LIBRARY



9 785845 910887