



OSBORNE



Содержит описание C99 – нового стандарта языка C, одобренного комитетами ANSI/ISO

Полный справочник

ПО

Рассмотрены
все
версии C

С Четвертое издание

"Герберт Шилдт рассказывает программистам об интересующей их теме доходчиво, кратко и авторитетно."

ACM Computing Reviews

Подробности
о языке
и библиотеках
функций

Рассмотрены все новейшие средства языка C, включая применение restrict к указателям, встроенные функции, массивы переменной длины, а также операции над комплексными числами и функции комплексного переменного

Сотни
примеров
и приложений

БЕСПЛАТНЫЙ
КОД
В
INTERNET
BONUS!
www.williamspublishing.com

Герберт Шилдт

Автор самых популярных книг по программированию. На его счету более 2,5 миллионов проданных книг!

Полный справочник по С

4-е издание

C: The Complete Reference,

Fourth Edition

Herbert Schildt

Osborne/**McGraw-Hill**

Berkeley New York St. Louis San Francisco

Auckland Bogota Hamburg London Madrid

Mexico City Milan Montreal New Delhi Panama City

Paris Sao Paulo Singapore Sydney

Tokyo Toronto

Полный справочник по С

4-е издание

Герберт Шилдт



Издательский дом "Вильямс"
Москва ♦ Санкт-Петербург ♦ Киев
2002

ББК 32.973.26-018.2.75

Ш55

УДК 681.3.07

Издательский дом “Вильямс”

Зав. редакцией *С.Н. Тригуб*

Перевод с английского *А.Г. Беляева, И.В. Константинова,
И.С. Литвинова, В.Д. Лузина, В.Н. Романова, А.Г. Сысолюка*

Под редакцией *Я.К. Шмидского*

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

Шилдт, Герберт.

Ш55 Полный справочник по C, 4-е издание. : Пер. с англ. — М. : Издательский дом “Вильямс”, 2002. — 704 с. : ил. — Парал. тит. англ.

ISBN 5-8459-0226-6 (рус.)

В данной книге, задуманной как справочник для программистов, работающих на языке C, подробно описаны все аспекты языка C и его библиотеки стандартных функций. Главный акцент сделан на стандарте ANSI/ISO языка C. Приведено описание как стандарта C89, так и C99. Особое внимание уделяется учету характеристик трансляторов, среды программирования и операционных систем, использующихся в настоящее время. Уже в самом начале подробно представлены все средства языка C, такие как ключевые слова, инструкции препроцессора и другие. Вначале описывается главным образом C89, а затем приводится подробное описание новых возможностей языка, введенных стандартом C99. Такая последовательность изложения позволяет облегчить практическое программирование на языке C, так как в настоящее время именно эта версия для большинства программистов представляется как “собственно C”, к тому же это самый распространенный в мире язык программирования. Кроме того, эта последовательность изложения облегчает освоение C++, который является надмножеством C89.

В книге много содержательных, нетривиальных примеров. Рассмотрены наиболее важные и распространенные алгоритмы и приложения, необходимые для каждого программиста, а также применение методов искусственного интеллекта и программирование для Windows 2000. Обсуждаются вопросы эффективности, переносимости и отладки программ. А в конце книги возможности языка C иллюстрируются на примере разработки его интерпретатора. Это, несомненно, самый лучший способ для осмысления, постижения и понимания чистоты и элегантности языка C.

ББК 32.973.26-018.2.75

Все названия программных продуктов являются зарегистрированными торговыми марками соответствующих фирм.

Никакая часть настоящего издания ни в каких целях не может быть воспроизведена в какой бы то ни было форме и какими бы то ни было средствами, будь то электронные или механические, включая фотокопирование и запись на магнитный носитель, если на это нет письменного разрешения издательства Osborne Publishing.

Authorized translation from the English language edition published by McGraw-Hill Companies, Copyright © 2000

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from the Publisher.

Russian language edition published by Williams Publishing House according to the Agreement with R&I Enterprises International, Copyright © 2001

ISBN 5-8459-0226-6 (рус.)

ISBN 0-07-212124-6 (англ.)

© Издательский дом “Вильямс”, 2001

© The McGraw-Hill Companies, 2000

Оглавление

Часть I. Основы языка C.....	29
Глава 1. Обзор возможностей языка C.....	31
Глава 2. Выражения.....	43
Глава 3. Операторы.....	81
Глава 4. Массивы и строки.....	107
Глава 5. Указатели.....	125
Глава 6. Функции.....	147
Глава 7. Структуры, объединения, перечисления и декларация typedef.....	169
Глава 8. Ввод/вывод на консоль.....	195
Глава 9. Файловый ввод/вывод.....	215
Глава 10. Препроцессор и комментарии.....	241
Часть II. Стандарт C99	253
Глава 11. C99.....	255
Часть III. Стандартная библиотека	273
Глава 12. Редактирование связей, использование библиотек и заголовков.....	275
Глава 13. Функции ввода/вывода.....	283
Глава 14. Строковые и символьные функции.....	319
Глава 15. Математические функции.....	345
Глава 16. Функции времени, даты и локализации.....	373
Глава 17. Функции динамического распределения памяти.....	385
Глава 18. Служебные функции.....	391
Глава 19. Функции обработки двухбайтовых символов.....	417
Глава 20. Библиотечные средства, добавленные в версии C99.....	425
Часть IV. Алгоритмы и приложения	435
Глава 21. Сортировка и поиск.....	437
Глава 22. Очереди, стеки, связанные списки и деревья.....	459
Глава 23. Разреженные массивы.....	493
Глава 24. Синтаксический разбор и вычисление выражений.....	509
Глава 25. Решение задач с помощью искусственного интеллекта.....	529
Часть V. Разработка программ с помощью C	571
Глава 26. Создание скелета приложения для Windows 2000.....	573
Глава 27. Проектирование программ с помощью C.....	591
Глава 28. Производительность, переносимость и отладка.....	603
Часть VI. Интерпретатор языка C	621
Глава 29. Интерпретатор языка C.....	623
Предметный указатель	681

Содержание

Предисловие	26
Часть I. Основы языка C.....	29
Глава 1. Обзор возможностей языка C.....	31
Краткая история развития языка C	32
C – язык среднего уровня	32
Язык C хорошо структурирован	34
Язык C создан для программистов	35
Компиляторы и интерпретаторы	36
Структура программы на языке C	37
Библиотеки и компоновка	38
Раздельная компиляция	39
Компиляция программы на языке C	39
Карта памяти программы на языке C	39
Сравнительная характеристика языков C и C++	40
Словарь терминов	41
Глава 2. Выражения	43
Базовые типы данных	44
Модификация базовых типов	45
Имена переменных	46
Переменные	47
Где объявляются переменные	47
Локальные переменные	47
Формальные параметры функции	50
Глобальные переменные	51
Четыре типа областей видимости	52
Квалификаторы типа	52
Квалификатор const	53
Квалификатор volatile	54
Спецификаторы класса памяти	54
Спецификатор extern	55
Спецификатор static	57
Спецификатор register	59
Инициализация переменных	60
Константы	60
Шестнадцатеричные и восьмеричные константы	61
Строковые константы	61
Специальные символьные константы	62
Операции	62
Оператор присваивания	63
Арифметические операции	65
Операции увеличения (инкремента) и уменьшения (декремента)	66
Операции сравнения и логические операции	67
Поразрядные операции	69
Операция ?	72
Операции получения адреса (&) и раскрытия ссылки (*)	73

Операция определения размера sizeof	74
Оператор последовательного вычисления: оператор “запятая”	75
Оператор доступа к члену структуры (оператор . (точка)) и оператор доступа через указатель -> (оператор стрелка)	75
Операторы [] и ()	76
Сводка приоритетов операций	76
Выражения	77
Порядок вычислений	77
Преобразование типов в выражениях	77
Явное преобразование типов: операция приведения типов	78
Пробелы и круглые скобки	79
Глава 3. Операторы	81
Логические значения ИСТИНА (True) и ЛОЖЬ (False) в языке C	82
Условные операторы	82
Оператор if	82
Вложенные условные операторы if	84
Лестница if-else-if	85
Оператор “?”, альтернативный условному	86
Условное выражение	88
Оператор выбора — switch	89
Вложенные операторы switch	92
Операторы цикла	92
Цикл for	92
Варианты цикла for	93
Бесконечный цикл	96
Цикл for без тела цикла	97
Объявление переменных внутри цикла	97
Цикл while	98
Цикл do-while	100
Операторы перехода	101
Оператор return	101
Оператор goto	101
Оператор break	102
Функция exit()	103
Оператор continue	104
Оператор-выражение	105
Блок операторов	105
Глава 4. Массивы и строки	107
Одномерные массивы	108
Создание указателя на массив	109
Передача одномерного массива в функцию	110
Строки	110
Двухмерные массивы	112
Массивы строк	115
Многомерные массивы	116
Индексация указателей	117
Инициализация массивов	118
Инициализация безразмерных массивов	120
Массивы переменной длины	121

Приемы использования массивов и строк на примере игры в крестики-нолики	121
Глава 5. Указатели.....	125
Что такое указатели.....	126
Указательные переменные.....	126
Операции для работы с указателями	127
Указательные выражения.....	127
Присваивание указателей	127
Преобразование типа указателя	128
Адресная арифметика	129
Сравнение указателей.....	130
Указатели и массивы.....	132
Массивы указателей.....	133
Многоуровневая адресация	134
Инициализация указателей	135
Указатели на функции	137
Функции динамического распределения.....	140
Динамическое выделение памяти для массивов.....	141
Указатели с квалификатором restrict.....	143
Трудности при работе с указателями	144
Глава 6. Функции.....	147
Общий вид функции.....	148
Что такое область действия функции	148
Аргументы функции.....	149
Вызовы по значению и по ссылке	149
Вызов по ссылке	150
Вызов функций с помощью массивов.....	152
Аргументы функции main(): argv и argc.....	154
Оператор return.....	157
Возврат из функции.....	157
Возврат значений.....	158
Возвращаемые указатели	160
Функции типа void.....	161
Что возвращает функция main()?	161
Рекурсия.....	161
Прототипы функций.....	163
Старомодные объявления функций.....	165
Прототипы стандартных библиотечных функций	166
Объявление списков параметров переменной длины.....	166
Правило “неявного int”	166
Старомодные и современные объявления параметров функций.....	167
Ключевое слово inline	168
Глава 7. Структуры, объединения, перечисления и декларация typedef.....	169
Структуры	170
Доступ к членам структуры.....	172
Присваивание структур	172
Массивы структур	173
Пример со списком рассылки.....	173
Передача структур функциям	179
Передача членов структур функциям	179
Передача целых структур функциям	179

Указатели на структуры	181
Объявление указателя на структуру	181
Использование указателей на структуры	181
Массивы и структуры внутри структур	183
Объединения	184
Битовые поля	187
Перечисления	188
Важное различие между C и C++	190
Использование sizeof для обеспечения переносимости	191
Средство typedef	192
Глава 8. Ввод/вывод на консоль	195
Чтение и запись символов	197
Трудности использования getchar()	197
Альтернативы getchar()	198
Чтение и запись строк	199
Форматный ввод/вывод на консоль	201
printf()	201
Вывод символов	202
Вывод чисел	202
Отображение адреса	204
Спецификатор преобразования %p	204
Модификаторы формата	205
Модификатор минимальной ширины поля	205
Модификатор точности	206
Выравнивание вывода	207
Обработка данных других типов	207
Модификаторы * и #	208
scanf()	208
Спецификаторы преобразования	209
Ввод чисел	209
Ввод целых значений без знака	210
Чтение одиночных символов с помощью scanf()	210
Чтение строк	210
Ввод адреса	211
Спецификатор %p	211
Использование набора сканируемых символов	211
Пропуск лишних разделителей	212
Символы в управляющей строке, не являющиеся разделителями	212
Функции scanf() необходимо передавать адреса	213
Модификаторы формата	213
Подавление ввода	214
Глава 9. Файловый ввод/вывод	215
Файловый ввод/вывод в C и C++	216
Файловый ввод/вывод в стандартном C и UNIX	216
Потоки и файлы	216
Потоки	217
Файлы	217
Основы файловой системы	218
Указатель файла	219
Открытие файла	219
Закрытие файла	220
Запись символа	221

Чтение символа	221
Использование fopen(), getc(), putc() и fclose()	222
Использование feof().....	223
Ввод/вывод строк: fputs() и fgets().....	224
Функция rewind()	225
Функция feerror()	226
Стирание файлов	227
Дозапись потока.....	228
Функции fread() и fwrite()	228
Использование fread() и fwrite().....	228
Пример со списком рассылки.....	229
Ввод/вывод при прямом доступе: функция fseek()	234
Функции fprintf() и fscanf()	235
Стандартные потоки	236
Связь с консольным вводом/выводом.....	238
Перенаправление стандартных потоков: функция freopen()	238
Глава 10. Препроцессор и комментарии	241
Препроцессор	242
Директива #define.....	242
Определение макросов с формальными параметрами	243
Директива #error.....	244
Директива #include.....	245
Директивы условной компиляции	245
Директивы #if, #else, #elif и #endif.....	245
Директивы #ifdef и #ifndef	247
Директива #undef	248
Использование defined	249
Директива #line	249
Директива #pragma.....	250
Операторы препроцессора # и ##.....	250
Имена предопределенных макрокоманд.....	251
Комментарии	251
Однострочные комментарии	252
Часть II. Стандарт C99	253
Глава 11. C99	255
Сравнение C99 с C89. Общее впечатление	256
Новые возможности.....	256
Удаленные средства	257
Измененные средства	257
Указатели, определенные с квалификаторами типа restrict.....	257
Ключевое слово inline	258
Новые встроенные типы данных	259
_Bool.....	259
_Complex и _Imaginary.....	260
Типы целых данных long long	260
Расширение массивов	260
Массивы переменной длины	261
Использование квалификаторов типов в объявлении массива.....	261
Однострочные комментарии	262
Распределение кода и объявлений	262
Изменения препроцессора.....	263

Переменные списки аргументов.....	263
Оператор <code>_Pragma</code>	263
Встроенные прагмы.....	263
Новые встроенные макросы.....	264
Объявление переменных внутри цикла <code>for</code>	264
Составные литералы.....	265
Массивы с переменными границами в качестве членов структур.....	266
Назначенные инициализаторы.....	266
Новые возможности семейства функций <code>printf()</code> и <code>scanf()</code>	267
Новые библиотеки C99.....	267
Зарезервированный идентификатор <code>__func__</code>	268
Расширение граничных значений трансляции.....	268
Неявный <code>int</code> больше не поддерживается	269
Удалены неявные объявления функций.....	270
Ограничения на <code>return</code>	270
Расширенные целые типы.....	270
Изменения в правилах продвижения целых типов.....	271
Часть III. Стандартная библиотека	273
Глава 12. Редактирование связей, использование библиотек и заголовков.....	275
Редактор связей	276
Раздельная компиляция	276
Переместимые коды и абсолютные коды.....	277
Редактирование связей с оверлеями.....	277
Связывание с динамически подключаемыми библиотеками (DLL).....	278
Стандартная библиотека C	279
Библиотечные файлы и объектные файлы.....	279
Заголовки.....	279
Макросы в заголовках	281
Переопределение библиотечных функций	281
Глава 13. Функции ввода/вывода.....	283
Функция <code>clearerr</code>	284
Пример	284
Зависимые функции	285
Функция <code>fclose</code>	285
Пример	286
Зависимые функции	286
Функция <code>feof</code>	286
Пример	286
Зависимые функции	286
Функция <code>ferror</code>	287
Пример	287
Зависимые функции	287
Функция <code>fflush</code>	287
Пример	287
Зависимые функции	288
Функция <code>fgetc</code>	288
Пример	288
Зависимые функции	288
Функция <code>fgetpos</code>	289
Пример	289
Зависимые функции	289

Функция fgets.....	289
Пример	289
Зависимые функции	290
Функция fopen.....	290
Пример	291
Зависимые функции	291
Функция fprintf.....	291
Пример	292
Зависимые функции	292
Функция fputc.....	292
Пример	293
Зависимые функции	293
Функция fputs.....	293
Пример	293
Зависимые функции	293
Функция fread.....	293
Пример	294
Зависимые функции	294
Функция freopen.....	294
Пример	295
Зависимые функции	295
Функция fscanf.....	295
Пример	296
Зависимые функции	296
Функция fseek.....	296
Пример	296
Зависимые функции	297
Функция fsetpos.....	297
Пример	297
Зависимые функции	297
Функция ftell.....	298
Пример	298
Зависимые функции	298
Функция fwrite.....	298
Пример	298
Зависимые функции	299
Функция getc.....	299
Пример	299
Зависимые функции	300
Функция getchar.....	300
Пример	300
Зависимые функции	300
Функция gets.....	300
Пример	301
Зависимые функции	301
Функция gettog.....	301
Пример	302
Функция printf.....	302
Модификаторы формата функции printf(), добавленные стандартом C99.....	304
Пример	305
Зависимые функции	305
Функция putc.....	305
Пример	305

Зависимые функции	305
Функция putchar.....	305
Пример	306
Зависимые функции	306
Функция puts	306
Пример	306
Зависимые функции	306
Функция remove	307
Пример	307
Зависимые функции	307
Функция rename	307
Пример	307
Зависимые функции	308
Функция rewind	308
Пример	308
Зависимые функции	308
Функция scanf.....	308
Модификаторы формата, добавленные к функции scanf() Стандартом C99	311
Пример	311
Зависимые функции	312
Функция setbuf	312
Пример	312
Зависимые функции	312
Функция setvbuf.....	312
Пример	313
Зависимые функции	313
Функция snprintf.....	313
Зависимые функции	313
Функция sprintf.....	313
Пример	314
Зависимые функции	314
Функция sscanf	314
Пример	314
Зависимые функции	315
Функция tmpfile.....	315
Пример	315
Зависимые функции	315
Функция tmpnam.....	315
Пример.....	316
Зависимые функции	316
Функция ungetc	316
Пример	316
Зависимые функции	317
Функции vprintf, fprintf, vsprintf и vsnprintf	317
Пример	317
Зависимые функции	318
Функции vscanf, fscanf и vsscanf.....	318
Зависимые функции	318
Глава 14. Строковые и символьные функции.....	319
Функция isalnum	320
Пример	320
Зависимые функции	321

Функция isalpha.....	321
Пример	321
Зависимые функции	321
Функция isblank.....	322
Пример	322
Зависимые функции	322
Функция iscntrl.....	322
Пример	322
Зависимые функции	323
Функция isdigit	323
Пример	323
Зависимые функции	323
Функция isgraph.....	324
Пример	324
Зависимые функции	324
Функция islower.....	324
Пример	324
Зависимые функции	325
Функция isprint.....	325
Пример	325
Зависимые функции	325
Функция ispunct	325
Пример	326
Зависимые функции	326
Функция isspace.....	326
Пример	326
Зависимые функции	327
Функция isupper	327
Пример	327
Зависимые функции	327
Функция isxdigit.....	327
Пример	328
Зависимые функции	328
Функция memchr.....	328
Пример	328
Еще один пример.....	329
Зависимые функции	329
Функция memchr	329
Пример	330
Зависимые функции	330
Функция memchr	330
Пример	331
Зависимые функции	331
Функция memmove	331
Пример	331
Зависимые функции	332
Функция memset.....	332
Пример	332
Зависимые функции	332
Функция strcat	332
Пример	332
Зависимые функции	333
Функция strchr.....	333

Пример	333
Зависимые функции	333
Функция <code>strcmp</code>	333
Пример	334
Зависимые функции	334
Функция <code>strcoll</code>	334
Пример	334
Зависимые функции	334
Функция <code>strcpy</code>	335
Пример	335
Зависимые функции	335
Функция <code>strncpy</code>	335
Пример	335
Зависимые функции	335
Функция <code>strcmp</code>	336
Пример	336
Функция <code>strlen</code>	336
Пример	336
Зависимые функции	336
Функция <code>strncat</code>	336
Пример	337
Зависимые функции	337
Функция <code>strncmp</code>	337
Пример	337
Зависимые функции	338
Функция <code>strncpy</code>	338
Пример	338
Зависимые функции	338
Функция <code>strbrk</code>	339
Пример	339
Зависимые функции	339
Функция <code>strchr</code>	339
Пример	339
Зависимые функции	340
Функция <code>strspn</code>	340
Пример	340
Зависимые функции	340
Функция <code>strstr</code>	340
Пример	340
Зависимые функции	341
Функция <code>strtok</code>	341
Пример	341
Зависимые функции	342
Функция <code>strxfrm</code>	342
Пример	342
Зависимые функции	342
Функция <code>tolower</code>	342
Пример	342
Зависимые функции	343
Функция <code>toupper</code>	343
Пример	343
Зависимые функции	343

Глава 15. Математические функции	345
Семейство функций acos	347
Пример	348
Зависимые функции	348
Семейство функций acosh	348
Зависимые функции	348
Семейство функций asin	348
Пример	349
Зависимые функции	349
Семейство функций asinh	349
Зависимые функции	349
Семейство функций atan	349
Пример	350
Зависимые функции	350
Семейство функций atanh	350
Зависимые функции	350
Семейство функций $\operatorname{atan2}$	350
Пример	351
Зависимые функции	351
Семейство функций cbrt	351
Пример	351
Зависимые функции	351
Семейство функций ceil	351
Пример	352
Зависимые функции	352
Семейство функций $\operatorname{copysign}$	352
Зависимые функции	352
Семейство функций cos	352
Пример	352
Зависимые функции	353
Семейство функций cosh	353
Пример	353
Зависимые функции	353
Семейство функций erf	353
Зависимые функции	354
Семейство функций erfc	354
Зависимые функции	354
Семейство функций exp	354
Пример	355
Зависимые функции	355
Семейство функций $\operatorname{exp2}$	355
Зависимые функции	355
Семейство функций $\operatorname{expm1}$	355
Зависимые функции	355
Семейство функций fabs	355
Пример	356
Зависимые функции	356
Семейство функций fdim	356
Зависимые функции	356
Семейство функций floor	356
Пример	356
Зависимые функции	357

Семейство функций fma	357
Зависимые функции	357
Семейство функций fmax	357
Зависимые функции	357
Семейство функций fmin	357
Зависимые функции	357
Семейство функций fmod	358
Пример	358
Зависимые функции	358
Семейство функций frexp	358
Пример	359
Зависимые функции	359
Семейство функций hypot	359
Зависимые функции	359
Семейство функций ilogb	359
Зависимые функции	359
Семейство функций ldexp	359
Пример	360
Зависимые функции	360
Семейство функций lgamma	360
Зависимые функции	360
Семейство функций llrint	360
Зависимые функции	360
Семейство функций llround	361
Зависимые функции	361
Семейство функций log	361
Пример	361
Зависимые функции	361
Семейство функций log1p	362
Зависимые функции	362
Семейство функций log10	362
Пример	362
Зависимые функции	362
Семейство функций log2	363
Зависимые функции	363
Семейство функций logb	363
Зависимые функции	363
Семейство функций lrint	363
Зависимые функции	363
Семейство функций lround	364
Зависимые функции	364
Семейство функций modf	364
Пример	364
Зависимые функции	364
Семейство функций nan	364
Зависимые функции	365
Семейство функций nearbyint	365
Зависимые функции	365
Семейство функций nextafter	365
Зависимые функции	365
Семейство функций nexttoward	365
Зависимые функции	365
Семейство функций pow	366

Пример	366
Зависимые функции	366
Семейство функций remainder	366
Зависимые функции	366
Семейство функций remquo	367
Зависимые функции	367
Семейство функций rint	367
Зависимые функции	367
Семейство функций round	367
Зависимые функции	367
Семейство функций scalbn	368
Зависимые функции	368
Семейство функций scalbn	368
Зависимые функции	368
Семейство функций sin	368
Пример	368
Зависимые функции	369
Семейство функций sinh	369
Пример	369
Зависимые функции	369
Семейство функций sqrt	370
Пример	370
Зависимые функции	370
Семейство функций tan	370
Пример	370
Зависимые функции	371
Семейство функций tanh	371
Пример	371
Зависимые функции	371
Семейство функций tgamma	371
Зависимые функции	371
Семейство функций trunc	372
Зависимые функции	372
Глава 16. Функции времени, даты и локализации.....	373
Функция asctime	374
Пример	374
Зависимые функции	375
Функция clock	375
Пример	375
Зависимые функции	375
Функция ctime	375
Пример	376
Зависимые функции	376
Функция difftime	376
Пример	376
Зависимые функции	376
Функция gmtime	377
Пример	377
Зависимые функции	377
Функция localeconv	377
Пример	379
Родственная функция	379

Функция localtime	379
Пример	380
Зависимые функции	380
Функция mktime	380
Пример	380
Зависимые функции	381
Функция setlocale	381
Пример	381
Зависимые функции	382
Функция strftime	382
Пример	383
Зависимые функции	384
Функция time	384
Пример	384
Зависимые функции	384
Глава 17. Функции динамического распределения памяти	385
Функция calloc	386
Пример	386
Зависимые функции	386
Функция free	387
Пример	387
Зависимые функции	387
Функция malloc	387
Пример	388
Зависимые функции	388
Функция realloc	388
Пример	389
Зависимые функции	389
Глава 18. Служебные функции	391
Функция abort	392
Пример	392
Зависимые функции	392
Функция abs	393
Пример	393
Зависимая функция	393
Функция-макрос assert	393
Пример	393
Зависимая функция	394
Функция atexit	394
Пример	394
Зависимые функции	394
Функция atof	394
Пример	395
Зависимые функции	395
Функция atoi	395
Пример	395
Зависимые функции	396
Функция atol	396
Пример	396
Зависимые функции	396
Функция atoll	396
Зависимые функции	396

Функция bsearch	397
Пример	397
Зависимая функция	398
Функция div	398
Пример	398
Зависимые функции	398
Функция exit	398
Пример	399
Зависимые функции	399
Функция _Exit	399
Зависимые функции	399
Функция getenv	399
Пример	400
Зависимая функция	400
Функция labs	400
Пример	400
Зависимые функции	400
Функция llabs	400
Зависимые функции	401
Функция ldiv	401
Пример	401
Зависимые функции	401
Функция lldiv	401
Зависимые функции	401
Функция longjmp	402
Пример	402
Зависимая функция	403
Функция mblen	403
Пример	403
Зависимые функции	403
Функция mbstowcs	403
Пример	403
Зависимые функции	403
Функция mbtowc	404
Пример	404
Зависимые функции	404
Функция qsort	404
Пример	405
Зависимая функция	405
Сортировка в убывающем порядке	405
Функция raise	405
Зависимая функция	406
Функция rand	406
Пример	406
Зависимая функция	406
Функция setjmp	406
Зависимая функция	407
Функция signal	407
Зависимая функция	407
Функция srand	407
Пример	408
Зависимая функция	408
Функция strtod	408

Пример	409
Зависимые функции	409
Функция strtouf.....	409
Зависимые функции	409
Функция strtol.....	410
Пример	410
Зависимые функции	410
Функция strtold.....	411
Зависимые функции	411
Функция strtoll.....	411
Зависимые функции	411
Функция strtoul.....	411
Пример	412
Зависимые функции	412
Функция strtoull.....	412
Зависимые функции	412
Функция system	413
Пример	413
Зависимая функция	413
Функции-макросы va_arg, va_start, va_end и va_copy.....	413
Пример	414
Зависимая функция	415
Функция wcstombs.....	415
Зависимые функции	415
Функция wctomb	415
Зависимые функции	415
Глава 19. Функции обработки двухбайтовых символов	417
Функции классификации двухбайтовых символов	418
Функции ввода-вывода двухбайтовых символов.....	420
Функции для операций над строками двухбайтовых символов.....	421
Преобразование строк двухбайтовых символов	422
Функции для обработки массивов двухбайтовых символов.....	423
Функции для преобразования многобайтовых и двухбайтовых символов	424
Глава 20. Библиотечные средства, добавленные в версии C99	425
Библиотека поддержки арифметических операций с комплексными числами...	426
Библиотека поддержки среды вычислений с плавающей точкой.....	430
Заголовок <stdint.h>	431
Функции для преобразования формата целочисленных значений.....	432
Математические макросы обобщенного типа	433
Заголовок <stdbool.h>.....	434
Часть IV. Алгоритмы и приложения	435
Глава 21. Сортировка и поиск	437
Сортировка	438
Классы алгоритмов сортировки.....	439
Оценка алгоритмов сортировки	439
Пузырьковая сортировка	440
Сортировка посредством выбора.....	443
Сортировка вставками	444
Улучшенные алгоритмы сортировки.....	445
Сортировка Шелла.....	446
Быстрая сортировка	448

Выбор метода сортировки.....	451
Сортировка других структур данных.....	451
Сортировка строк.....	451
Сортировка структур.....	453
Сортировка дисковых файлов с произвольной выборкой.....	454
Поиск.....	457
Методы поиска.....	457
Последовательный поиск.....	457
Двоичный поиск.....	458
Глава 22. Очереди, стеки, связанные списки и деревья.....	459
Очереди.....	460
Циклическая очередь.....	464
Стеки.....	467
Связанные списки.....	471
Односвязные списки.....	471
Двусвязные списки.....	476
Пример списка рассылки.....	479
Двоичные деревья.....	484
Глава 23. Разреженные массивы.....	493
Зачем нужны разреженные массивы?.....	494
Представление разреженного массива в виде связанного списка.....	495
Анализ метода представления в виде связанного списка.....	498
Представление разреженного массива в виде двоичного дерева.....	498
Анализ метода представления в виде двоичного дерева.....	500
Представление разреженного массива в виде массива указателей.....	500
Анализ метода представления разреженного массива в виде массива указателей.....	502
Хэширование.....	503
Анализ метода хэширования.....	506
Выбор метода.....	507
Глава 24. Синтаксический разбор и вычисление выражений.....	509
Выражения.....	510
Разбиение выражения на лексемы.....	511
Разбор выражений.....	514
Простая программа синтаксического анализа выражений.....	515
Работа с переменными в анализаторе.....	520
Проверка синтаксиса в рекурсивном нисходящем анализаторе.....	526
Глава 25. Решение задач с помощью искусственного интеллекта.....	529
Представление и терминология.....	530
Комбинаторные взрывы.....	532
Методы поиска.....	534
Оценка поиска.....	535
Представление в виде графа.....	536
Поиск в глубину.....	537
Анализ поиска в глубину.....	545
Полный перебор, или поиск в ширину.....	546
Анализ поиска в ширину.....	547
Добавление эвристики.....	548
Поиск методом наискорейшего подъема.....	548
Анализ наискорейшего подъема.....	554
Поиск с использованием частичного пути минимальной стоимости.....	554

Анализ поиска с использованием частичного пути минимальной стоимости .	555
Выбор метода поиска	556
Поиск нескольких решений	556
Удаление путей	557
Удаление вершин	557
Поиск “оптимального” решения	562
И снова возвращаемся к поиску потерянных ключей.....	567
Часть V. Разработка программ с помощью С	571
Глава 26. Создание скелета приложения для Windows 2000	573
Общая картина специфики программирования для Windows 2000	574
Модель рабочего стола	575
Мышь.....	575
Пиктограммы, растровые изображения и другая графика	575
Меню, средства управления и диалоговые окна.....	576
Интерфейс прикладного программирования Win32	576
Компоненты окна	577
Взаимодействие прикладных программ с Windows.....	578
Базовые концепции функционирования приложений для Windows 2000.....	578
WinMain()	579
Процедура окна.....	579
Классы окон	579
Цикл обработки сообщений.....	580
Типы данных Windows.....	580
Скелет программы для Windows 2000	580
Определение класса окна	583
Создание окна	585
Цикл обработки сообщений.....	587
Функция окна	588
Файл описания больше не нужен	589
Соглашения об именовании	589
Глава 27. Проектирование программ с помощью С.....	591
Проектирование сверху вниз.....	592
Структурирование программы	592
Выбор структуры данных	593
“Пуленепробиваемые” функции	594
Использование программы MAKE	597
Использование макросов в MAKE	600
Применение интегрированной среды разработки	601
Глава 28. Производительность, переносимость и отладка	603
Эффективность.....	604
Операции увеличения и уменьшения.....	604
Применение регистровых переменных	605
Указатели вместо индексации массива.....	606
Применение функций	606
Перенос программ.....	610
Использование #define	610
Зависимость от операционной системы.....	611
Различия в размерах данных	611
Отладка	611
Ошибки очередности вычисления	611

Проблемы с указателями.....	612
Интерпретация синтаксических ошибок.....	614
Ошибки, вызванные “потерей” единицы	615
Ошибки из-за нарушения границ.....	616
Пропуск прототипов функций.....	617
Ошибки при задании аргументов	618
Переполнение стека.....	619
Применение отладчика.....	619
Теория отладки в общих чертах.....	619
Часть VI. Интерпретатор языка C	621
Глава 29. Интерпретатор языка C.....	623
Практическое значение интерпретаторов.....	624
Определение языка Little C	625
Ограничения языка Little C	626
Интерпретация структурированного языка	628
Неформальная теория языка C	628
Выражения языка C.....	629
Определение значения выражения	630
Синтаксический анализатор выражений	631
Синтаксический разбор исходного текста программы.....	631
Рекурсивный нисходящий синтаксический анализатор Little C	636
Интерпретатор Little C.....	647
Предварительный проход интерпретатора	648
Функция main().....	650
Функция interp_block()	651
Обработка локальных переменных	664
Вызов функций, определенных пользователем	665
Присваивание значений переменным	668
Выполнение оператора if.....	669
Обработка цикла while.....	670
Обработка цикла do-while	671
Цикл for.....	671
Библиотечные функции Little C.....	672
Компиляция и компоновка интерпретатора Little C.....	675
Демонстрация Little C.....	675
Усовершенствование интерпретатора Little C.....	679
Расширение Little C	680
Добавление новых средств в язык Little C.....	680
Создание дополнительных средств программирования	680
Предметный указатель	681

Об авторе

Герберт Шилдт (Herbert Schildt) — выдающийся автор книг по программированию, общепризнанный авторитет в области программирования на языках C, C++, Java и приложений для Windows. Тираж его книг, переведенных на многие языки мира, составляет более 2,5 миллионов экземпляров. Он является автором многочисленных бестселлеров, среди которых наиболее известны такие издания, как *C++: The Complete Reference*, *Teach Yourself C*, *Teach Yourself C++*, *C++ from the Ground Up*, *Windows 2000 Programming from the Ground Up*, *Java: The Complete Reference* (Полный справочник по Java, изд. «Диалектика»). Г. Шилдт имеет степень магистра наук в области вычислительной математики, присвоенную ему Иллинойским университетом (University of Illinois).

Предисловие

Э то четвертое издание книги *C: The Complete Reference (Полный справочник по C)*. Со времен третьего издания в области программирования произошло много прогрессивных изменений, в частности, получили широкое распространение Internet и World Wide Web, был изобретен язык Java, а C++ был стандартизирован. Также был создан новый стандарт C, названный C99. Несмотря на то, что Стандарт C99 пока не завоевал всеобщего признания, его создание стало одним из самых выдающихся событий в области программирования за последние пять лет. В условиях стремительного развития компьютерных технологий порой нелегко сразу определить фундаментальные элементы, на которых строится будущее этих технологий. Именно таким основополагающим элементом является язык C. Во всем мире значительная часть текстов программ написана на этом языке. На его основе построен язык C++, а синтаксис языка C является фундаментом языка Java. Если бы язык C был всего лишь отправной точкой для других языков программирования, это был бы интересный, но мертвый язык. К счастью, это не так. Сегодня язык C не менее актуален, чем во время своего создания. Как будет видно из дальнейшего изложения, Стандарт C99 содержит такие новые перспективные конструкции, благодаря которым язык C по праву считается одним из самых прогрессивных в области программирования. Несмотря на большое распространение “потомков” языка C (Java и C++), значение самого C по-прежнему остается первостепенным.

В создании Стандарта C99 участвовали наиболее выдающиеся специалисты по языкам программирования, среди них такие, как Рекс Джашке (Rex Jaeschke), Джим Томас (Jim Thomas), Том МакДональд (Tom MacDonald) и Джон Бенито (John Benito). Как член комиссии по стандартизации я наблюдал процесс создания стандарта и участвовал в дискуссиях по поводу каждого нововведения. В результате ежедневного обмена идеями и мнениями по электронной почте между участниками этого процесса, находящимися практически во всех странах мира, удалось выработать единую концепцию, что и привело в конечном итоге к значительному усовершенствованию языка C.

Надо признать, что, работая над первым изданием этой книги, я не предполагал столь быстрого роста достижений в области программирования, хотя некоторые из них, например, большой успех C++, были предопределены уже с самого начала. Но язык C я считаю одним из лучших среди всех языков программирования. Это изящный, элегантный, логичный и, что особенно важно, мощный язык. Его столь успешное развитие и распространение очень меня радует и вдохновляет.



Книга для всех программистов

Эта книга задумана как справочник для всех программистов, работающих на языке C, независимо от уровня их подготовки. Предполагается, что читатель уже имеет некоторое представление об основах языка C и может написать на нем хотя бы простейшую программу. Однако, если читатель только начал изучать C, эта книга послужит отличным дополнением к любому учебнику по C, так как в ней можно будет найти ответы на многие трудные вопросы, возникающие в процессе изучения.

Книга также будет полезна в качестве подробного справочника по основам C++, который, как известно, является объектно-ориентированным расширением языка C, т.е. она пригодится любому программисту, пишущему программы на C или C++.



Новое в четвертом издании

По сравнению с тремя предыдущими изданиями структура книги в основном осталась неизменной. Большинство изменений определено новыми возможностями языка, появившимися после введения Стандарта С99. Все эти новые возможности подробно описаны в части II книги, в предыдущих изданиях эта информация отсутствует. Часть III, в которой описывается библиотека стандартных функций, в этом издании значительно дополнена, в нее включено описание многих новых библиотечных функций, введенных Стандартом С99. Но, конечно, не изменено полное описание Стандарта С89, который, как известно, очень важен, ведь именно на его основе создан С++. Кроме того, большинство программистов работают с версией С89, потому что до настоящего времени фактически все еще нет общедоступного компилятора, поддерживающего Стандарт С99.

Книга в целом была значительно переработана с целью ознакомления с новыми характеристиками трансляторов, среды программирования и операционных систем, использующихся в настоящее время.



О чем эта книга

В книге подробно описаны все аспекты языка С и его библиотеки стандартных функций. Главный акцент сделан на Стандарте ANSI/ISO этого языка. Дано описание как Стандарта С89, так и С99.

Книга состоит из следующих шести частей:

- Основопологающие элементы языка С, определенные в С89
- Расширение С99
- Библиотеки стандартных функций С
- Распространенные алгоритмы и приложения
- Среда программирования С
- Создание интерпретатора С

В части I подробно представлены все средства языка С, т.е. его ключевые слова, инструкции препроцессора и другие. В этой части в основном описывается Стандарт С89, а также упоминаются некоторые новые свойства, введенные Стандартом С99.

В части II подробно рассматриваются новые возможности языка, введенные стандартом С99. Есть два аргумента в пользу отдельного описания стандартов С89 и С99. Во-первых, подавляющее большинство программистов используют сегодня С89. Эта версия языка С воспринимается ими как "собственно С". К тому же это самый распространенный в мире язык программирования. Существенно также и то, что С89 является подмножеством С++. Поэтому версия С89 крайне актуальна как сегодня, так и в обозримом будущем. По этим причинам в книге должно быть сделано четкое разграничение между этими версиями языка С. Во-вторых, многие читатели этой книги уже хорошо знакомы с версией С89, и им будет значительно легче найти новый для себя материал, если новые свойства С99 будут изложены в отдельной части книги.

В части III дается описание библиотеки стандартных функций С. Рассматриваются как функции Стандарта С89, так и функции Стандарта С99, причем особо выделены функции, введенные Стандартом С99.

В части IV можно ознакомиться с наиболее важными и распространенными алгоритмами и приложениями, необходимыми для каждого программиста. Здесь рассматриваются методы искусственного интеллекта и их применение, а также программирование для Windows 2000.

В части V вы узнаете много интересного о среде программирования C, здесь обсуждаются вопросы эффективности, переносимости и отладки программ.

В части VI возможности языка C демонстрируются на примере разработки его интерпретатора. Это наиболее увлекательная и даже забавная часть книги. Поэкспериментировать с этим интерпретатором будет истинным наслаждением для любого программиста! Нам кажется, это самый лучший способ для того, чтобы по достоинству оценить чистоту и элегантность языка C.



Тексты программ в Web

Тексты программ, рассматриваемых в этой книге, можно бесплатно получить по адресу www.williamspublishing.com

HS

21 марта 2000 г.

Магомет (Mahomet), штат Иллинойс



Материал для дальнейшего изучения

Эта книга Герберта Шилдта — лишь одна из многих его книг по программированию. Ниже приведены другие книги, представляющие интерес для программистов.

Изучающим программирование под Windows мы рекомендуем следующие книги:

Windows 2000 Programming from the Ground Up

Windows 98 Programming from the Ground Up

Windows NT 4 Programming from the Ground Up

The Windows Programming Annotated Archives

Для изучающих язык C будут интересны такие книги:

Teach Yourself C

C/C++ Annotated Archives

Следующие книги будут полезны тем, кто изучает C++:

C++: The Complete Reference

Teach Yourself C++

C++ from the Ground Up

Expert C++

C/C++ Annotated Archives

Изучающим язык Java мы рекомендуем прочесть:

Java: The Complete Reference (Полный справочник по Java, изд. "Диалектика").



Если вам нужна квалифицированная консультация, обращайтесь к Герберту Шилдту, общепризнанному авторитету в области программирования

Полный справочник по



Часть I

Основы языка C

Описание языка C в данной книге разделено на две части. В части I рассматриваются свойства C, определенные Стандартом ANSI 1989 года (Стандарт C89) с учетом 1-й Поправки, принятой в 1995 году. В настоящее время эта версия C широко распространена и поддерживается всеми существующими компиляторами C. Эта версия является также основой языка C++, поэтому на нее обычно ссылаются как на подмножество C++. В части II описываются новые свойства C,

введенные Стандартом 1999 года (C99); здесь же дано подробное объяснение, чем C99 отличается от C89. Новый стандарт 1999 года почти полностью базируется на Стандарте 1989 года, появились лишь некоторые новые возможности языка, принципиально не повлиявшие на его суть. Таким образом, C89 является основой как C99, так и C++.

Отдельное рассмотрение языка C в двух аспектах — C89 как основы и специфических свойств C99 — имеет три главных преимущества:

- Достаточно четко определены различия между C89 и C99. В настоящее время пока еще нет общедоступных компиляторов C99, поэтому для программиста очень важно понимание этих различий. В противном случае может оказаться, что, решая определенную задачу, программист рассчитывает на средства языка, не поддерживаемые ни одним из существующих компиляторов.
- Многие читатели, знакомые с C89, без труда обнаружат новые свойства языка, введенные Стандартом C99. Изложение версии C99 в отдельной части облегчает для квалифицированных программистов задачу поиска новой информации о версии C99. Но следует отметить, что и в части I там, где это уместно, упоминаются отличия C89 от C99 и новые возможности языка, введенные Стандартом C99.
- Отдельное рассмотрение версии C89 позволяет предельно четко определить версию C, образующую подмножество C++. Это особенно важно при написании тех программ на C, которые будут транслироваться компилятором C++. Это необходимо также в том случае, если планируется переход на компилятор C++ или возникает необходимость работать с обоими компиляторами одновременно.

Знание различий между C89 и C99 является обязательным для каждого высококвалифицированного специалиста C, работающего с программами на языке C.

Часть I построена следующим образом:

- Глава 1 — обзор возможностей языка C;
- Глава 2 — рассматриваются базовые типы данных, переменные, операторы и выражения;
- Глава 3 — рассматриваются управляющие операторы программы;
- Глава 4 — обсуждаются массивы и строки;
- Глава 5 — описание указателей;
- Глава 6 — рассматриваются функции;
- Глава 7 — описание структур, объединений и пользовательских типов данных.
- Глава 8 — рассматривается консольный ввод/вывод данных;
- Глава 9 — рассматривается ввод и вывод данных в файлы;
- Глава 10 — описание препроцессора и комментарии.

Полный
справочник по



Глава 1

Обзор возможностей языка C

Целью этой главы является описание возможностей языка программирования C, ознакомление с его происхождением, использованием, а также лежащими в его основе концепциями. Глава рассчитана главным образом на программистов, которые только приступили к изучению языка C.

Краткая история развития языка C

Язык C был изобретен и реализован Деннисом Ритчи (Dennis Ritchie) для компьютера DEC PDP-11 в операционной системе Unix. Этот язык был разработан на основе “более старого” языка BCPL, созданного в свое время Мартином Ричардсом (Martin Richards). BCPL оказал определенное влияние на язык B, разработанный Кеном Томпсоном (Ken Thompson). В свою очередь развитие языка B привело к созданию в 1970 году языка C.

На протяжении многих лет стандартом C была фактически версия, поставляемая вместе с операционной системой Unix. Эта версия впервые была описана Брайаном Керниганом (Brian Kernighan) и Деннисом Ритчи в книге *The C Programming Language* (Englewood Cliffs, N.J.: Prentice-Hall, 1978). Летом 1983 года был образован комитет по созданию для языка C стандарта ANSI (*American National Standards Institute — Национальный институт стандартизации США*). Надо отметить, что процесс стандартизации занял весьма немалый срок — шесть лет.

Стандарт ANSI был окончательно принят в декабре 1989 года и впервые опубликован в начале 1990 года. Этот стандарт был также принят организацией ISO (*International Standards Organization — Международная организация по стандартизации*), поэтому он называется стандартом ANSI/ISO языка C. В 1995 году была принята 1-я Поправка к стандарту C, согласно которой, среди прочего, было добавлено несколько библиотечных функций. В 1989 году стандарт C вместе с 1-й Поправкой стал *базовым документом* Стандарта C++, определяющего C как подмножество C++. Версию C, определенную стандартом 1989 года, обычно называют C89.

На протяжении 90-х годов внимание программистов было приковано главным образом к развитию стандарта C++. Тем временем разработка C также продолжалась, приведя в 1999 году к появлению стандарта C, который принято называть C99. В целом C99 сохранил все основные черты C89, т.е., можно сказать, что язык C остался самим собой! Комиссия стандартизации C99 уделила основное внимание двум направлениям: добавлению нескольких численных библиотек и развитию новых узкоспециальных средств, таких как массивы переменной длины и модификатор указателя `restrict`. Благодаря этим нововведениям язык C опять оказался на переднем крае развития языков программирования.

Как указано выше, часть I этой книги посвящена основам C в версии стандарта 1989 года. В настоящее время это наиболее распространённая версия, она используется всеми компиляторами C, она же является основой для C++. Поэтому при написании программ на C для любого из существующих в настоящее время компиляторов необходимо придерживаться описания, приведенного в части I книги. В части II рассматриваются свойства, введенные стандартом C99.

C — язык среднего уровня

Язык C часто называют языком программирования среднего уровня. Но это не значит, что C менее мощный, менее развитой или более трудный в использовании, чем языки высокого уровня, такие как Basic или Pascal. Это также не значит, что C такой же громоздкий и неудобный, как ассемблер. Языком среднего уровня его назы-

вают скорее потому, что он объединяет в себе лучшие черты языков высокого уровня с возможностями ассемблера. В табл. 1.1 показано, какое место занимает С среди других языков программирования.

Таблица 1.1. Место языка С среди других языков программирования

Языки высокого уровня	Ada Modula-2 Pascal COBOL FORTRAN Basic
Языки среднего уровня	Java C++ C FORTH
Языки низкого уровня	Макроассемблер Ассемблер

Как язык среднего уровня С позволяет манипулировать битами, байтами и адресами, то есть теми базовыми элементами данных, с которыми работает компьютер. Несмотря на это программа, написанная на С, обладает высокой переносимостью. *Переносимость* — это свойство программного обеспечения, созданного для одного типа компьютера или операционной системы, позволяющее легко переделать его для другого типа, т.е. перенести в другую вычислительную среду. Например, если программу, работающую под управлением DOS, легко переделать так, чтобы она работала под управлением Windows 2000, то такая программа называется *переносимой*¹.

Все языки высокого уровня придерживаются концепции *типов данных*. Тип данных представляет собой набор значений, хранящихся в переменных, а также набор операций, выполнение которых допускается над этими значениями. Обычные типы данных — это целые числа, символы и числа с плавающей точкой. Язык С имеет несколько встроенных типов данных, однако он не является сильно типизированным языком, как Pascal или Ada. В языке С допускаются почти все преобразования типов. Например, в выражениях можно свободно смешивать переменные символьного и целого типов.

В отличие от большинства языков высокого уровня, в С почти отсутствует контроль ошибок в процессе выполнения программы. Например, не проверяется нарушение границ массивов. Ответственность за подобные ошибки полностью возлагается на программиста.

Аналогично этому С не требует строгой совместимости параметров и аргументов функций. В языках программирования высокого уровня обычно необходимо, чтобы тип аргумента более или менее соответствовал типу параметра. Для С это не характерно, здесь аргумент может иметь почти любой тип, если его можно разумно преобразовать в тип параметра. Более того, компилятор С автоматически осуществляет все виды необходимых преобразований.

Отличительной особенностью языка С является возможность манипулирования непосредственно битами, байтами, словами и указателями. Поэтому С хорошо приспособлен для системного программирования.

Другая важная особенность С — это малое количество ключевых слов, составляющих команды языка. В С89 определено 32 ключевых слова, причем в С99 добавлено только 5 слов. Языки высокого уровня обычно имеют значительно больше ключевых слов, например, в большинстве версий языка Basic их количество превышает сотню!

¹ А также машиннезависимой, мобильной, а иногда даже портативной. — Прим. ред.



Язык C хорошо структурирован

В книгах по программированию часто используется понятие *блочной структурированности* языка. Хотя этот термин и нельзя применить в полной мере к языку C, его обычно называют просто *структурированным* языком, так как в этом отношении он очень похож на другие структурированные языки, такие как ALGOL, Pascal и Modula-2.

На заметку

Блочно-структурированные языки допускают определение функций внутри других функций. Поскольку в C такой возможности нет, формально он не может быть причислен к блочно-структурированным языкам.

Отличительной особенностью структурированного языка является *отдельное размещение* различных частей кода программы и данных. Таким способом программист может “скрыть” часть информации, используемую для выполнения специфической задачи, от тех участков программы, где эта информация не нужна. Один из способов достижения этого — использование подпрограмм с локальными переменными. В этом случае любые действия внутри программы не вызовут побочных эффектов в других ее частях. Это позволяет программам, написанным на C, совместно использовать готовые части кода. Для использования функции, хранящейся отдельно, необходимо только знать, что эта функция делает, при этом вовсе не обязательно знать, как именно она это делает. Но следует помнить, что чрезмерное использование глобальных переменных (то есть переменных, видимых во всей программе) приводит к ошибкам и побочным эффектам, которые очень трудно устранить (особенно хорошо знакомы с этой трудностью программисты, работавшие на стандартной версии языка Basic).

Структурированный язык предоставляет программисту много различных возможностей. Например, структурированные языки обычно содержат несколько типов операторов цикла, таких как while, do-while и for. В структурированных языках использование оператора goto или запрещено, или не рекомендуется, для них он не является приемлемым средством управления процессом (что, однако, не относится к стандартной версии языка Basic и традиционной версии языка FORTRAN). Структурированный язык позволяет поместить оператор в любом месте строки, не привязывая его к определенному полю (что характерно, например, для старых версий языка FORTRAN).

Ниже приведены примеры структурированных и неструктурированных языков:

Неструктурированные	Структурированные
FORTRAN	Pascal
BASIC	ADA
COBOL	C++
	C
	Java
	Modula-2

Структурированные языки появились сравнительно недавно. Фактически признаком того, что язык создан довольно давно, служит его неструктурированность. Сегодня мало кто из программистов решится писать серьезную программу на неструктурированном языке.

На заметку

Попытки добавить элементы структурированности во многие старые языки предпринимались неоднократно. Так, одна из самых смелых попыток была предпринята для языка Basic. Однако преодолеть недостатки этих языков не удалось, так как при их создании изначально были проигнорированы принципы структурированности.

Главная конструкция структурного программирования на языке C — функция, являющаяся здесь единственным видом подпрограммы. Функция C — это строительный

кирпичик, в котором осуществляются все действия программы. Функции позволяют определить и отдельно закодировать различные задачи, решаемые программой, благодаря чему эта программа становится модульной. Написав правильно функцию, можно быть уверенным в ее надежной работе в различных ситуациях без побочных эффектов в других частях программы. При работе над большим проектом, когда особенно важно, чтобы одна часть кода ни в коем случае не могла непредвиденно подействовать на другую часть, умение создать отдельную функцию приобретает для программиста исключительное значение.

Другой способ структурирования программы, написанной на языке C, заключается в использовании программных блоков. *Программный блок* — это логически связанная группа операторов программы, которую можно рассматривать как отдельную программную единицу. В языке C блок представляет собой последовательность операторов программы, заключенную в фигурные скобки. В примере кода

```
if (x<10) {  
    printf("Слишком мало, попробуйте еще раз.\n");  
    scanf("%d", &x);  
}
```

два оператора, стоящие после if в фигурных скобках, выполняются только в том случае, если значение x меньше десяти. Эти два оператора вместе со скобками составляют программный блок. В данном примере эти операторы представляют собой логический блок, или программную единицу, так как один оператор не может быть выполнен без выполнения другого. Использование программных блоков позволяет сделать программу понятной, элегантной и эффективной. Более того, программные блоки помогают лучше формализовать задачу и более точно запрограммировать алгоритм ее решения.

Язык C создан для программистов

Как ни удивительно, но не все языки программирования созданы для программистов. Классические примеры языков для непрограммистов — COBOL и Basic. COBOL был создан не для того, чтобы облегчить жизнь программистам или повысить надежность программного продукта, и даже не для повышения продуктивности труда программиста, а для того, чтобы непрограммисты могли читать и понимать написанные на нем программы. При создании языка Basic в значительной степени преследовалась цель сделать для непрограммиста доступным решение на компьютере относительно простых задач.

В противовес этому язык C был создан и апробирован активно работающими программистами. В результате C обеспечивает то, чего и ждут от него именно программисты: небольшое количество ограничений, блочную структуру, автономные функции и малое количество ключевых слов. Программы, написанные на языке C, обладают эффективностью программ, написанных на языке ассемблера, и структурированностью, присущей программам, созданным на языках Pascal или Modula-2. Поэтому неудивительно, что во всем мире C стал универсальным языком программирования.

Решающим фактором успеха языка C стало то, что во многих случаях он может быть использован вместо ассемблера, который основан на символическом представлении бинарного кода, непосредственно выполняемого компьютером. Каждая операция ассемблера представляет для компьютера одну элементарную задачу. Разрабатывая программу на языке ассемблера, программист может сделать программу максимально гибкой и эффективной, однако работа с самой программой ассемблера и ее отладка — чрезвычайно трудоемкий процесс. Более того, из-за отсутствия средств структурного программирования в языке ассемблера окончательная программа представляет собой то, что программисты называют “спагетти” — хаотичную совокупность переходов, индексов и вызовов функций. Из-за своей неструктурированности программа, написанная на языке ассемблера, с большим

трудом поддается расширению, модификации и даже просто пониманию. И что, возможно, наиболее существенно, процедуры, написанные на языке ассемблера, не обладают переносимостью на компьютеры с процессорами, система команд которых отличается от системы команд исходного процессора.

Первоначально С использовался для решения задач системного программирования. *Системная программа* — это часть операционной системы компьютера или утилиты, как, например, редактор, транслятор, компоновщик и т.п. По мере роста популярности С, многие программисты стали использовать его для решения других задач благодаря его переносимости и эффективности, а также потому, что им это нравилось! Поистине этот язык стал долгожданным и впечатляющим достижением в области языков программирования.

С появлением языка С++ многим программистам стало казаться, что С прекратил свое существование как отдельный язык программирования. Однако это не так. Во-первых, не для всех программ нужны объектно-ориентированные возможности С++. Например, такие приложения как системы внедрения объектов, по-прежнему программируются главным образом на С. Во-вторых, в настоящее время во всем мире работает чрезвычайно много программ, написанных на С, причем разработчики продолжают модернизировать и поддерживать эти программы. В-третьих, разработка нового стандарта С99 убеждает в том, что развитие и совершенствование С продолжается. То, что С стал базисом для С++, навсегда останется его неоспоримой заслугой, и в то же время язык С сам остается одним из лучших языков программирования.



Компиляторы и интерпретаторы

Программист должен понимать, что язык программирования определяет характер программы, а не способ ее выполнения компьютером. Есть два способа выполнения программы компьютером: она может быть подвергнута *компиляции* или *интерпретации*. Программа, написанная на любом языке программирования, может как компилироваться, так и интерпретироваться, однако многие языки изначально созданы для выполнения преимущественно одним из этих способов. Например, Java рассчитан в основном на интерпретацию программы, а язык С — на компиляцию. Необходимо помнить, что при разработке языка С его конструкции оптимизировались специально для компиляции. И хотя интерпретаторы С существуют и доступны для программистов (особенно как средства отладки или объект для экспериментов — в качестве такого объекта можно использовать, например, интерпретатор, рассмотренный в части VI этой книги), С разрабатывался преимущественно для компиляции. Поэтому при разработке программ на С большинство программистов используют именно компилятор, а не интерпретатор. Поскольку не все читатели этой книги достаточно хорошо понимают отличие компилятора от интерпретатора, ниже дается краткое разъяснение по этому поводу.

В простейшем случае интерпретатор читает исходный текст программы по одной строке за раз, выполняет эту строку и только после этого переходит к следующей. Так работали ранние версии языка Basic. В языках типа Java исходный текст программы сначала конвертируется в промежуточную форму, а затем интерпретируется. В этом случае программа также интерпретируется в процессе выполнения.

Компилятор читает сразу всю программу и конвертирует ее в *объектный код*, то есть транслирует исходный текст программы в форму, более пригодную для непосредственного выполнения компьютером. Объектный код также называют *двоичным* или *машинным кодом*. Когда программа скомпилирована, в ее коде уже нет отдельных строк исходного кода.

В общем случае интерпретируемая программа выполняется медленнее, чем скомпилированная. Необходимо помнить, что компилятор преобразует исходный текст

программы в объектный код, который выполняется компьютером непосредственно. Значит, потеря времени на компиляцию происходит лишь единожды, а в случае интерпретации — каждый раз при очередной компиляции фрагмента программы в процессе ее выполнения.

Структура программы на языке C

В табл. 1.2 перечислены 32 ключевых слова, определенные стандартом C89. Они же являются ключевыми словами языка C как подмножества C++. В табл. 1.3 приведены ключевые слова, добавленные стандартом C99. Набор ключевых слов вместе с формальным синтаксисом C составляет язык программирования C.

Кроме стандартных ключевых слов, многие компиляторы для лучшего функционирования в среде программирования разрешают дополнительно использовать некоторые нестандартные ключевые слова. Например, несколько компиляторов, рассчитанных на создание кода, выполняемого в моделях памяти, поддерживаемых процессорами семейства 8086, с целью поддержки взаимодействия программ, написанных на разных языках, а также для обеспечения доступа к прерываниям дополнительно вводят следующие ключевые слова:

asm	_ds	huge	pascal
cdecl	_es	interrupt	_ss
_cs	far	near	

Для наиболее эффективного использования возможностей конкретного компилятора программист обязательно должен ознакомиться с набором дополнительных ключевых слов.

Таблица 1.2. Ключевые слова стандарта C89

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

Таблица 1.3. Ключевые слова, добавленные стандартом C99

_Bool	_Imaginary	restrict
_Complex	inline	

В языке C различаются верхний и нижний регистры символов: `else` — ключевое слово, а `ELSE` — нет. В программе ключевое слово может быть использовано только как ключевое слово, то есть никогда не допускается его использование в качестве переменной или имени функции.

Любая программа на C состоит из одной или нескольких функций. Обязательно должна быть определена единственная главная функция `main()`, именно с нее всегда начинается выполнение программы. В хорошем исходном тексте программы главная функция всегда содержит операторы, отражающие сущность решаемой задачи, чаще всего это вызовы функций. Хотя `main()` и не является ключевым словом, относиться

к нему следует как к ключевому. Например, не следует использовать `main` как имя переменной, так как это может нарушить работу транслятора.

Структура программы C изображена на рис. 1.1, здесь `f1()` — `fN()` означают функции, написанные программистом.

```
Объявление глобальных переменных
int main(список параметров)
{
    последовательность операторов
}
тип_возвращаемого_значения f1(список п
{
    последовательность операторов
}
тип_возвращаемого_значения f2(список п
{
    последовательность операторов
}
.
.
.
тип_возвращаемого_значения fN(список п
{
    последовательность операторов
}
}
```

Рис. 1.1. Структура программы на языке C

Библиотеки и компоновка

Следует отметить, что на C в принципе возможно создать программу, содержащую только имена переменных и ключевые слова. Но обычно так не поступают, потому что в C нет ключевых слов для выполнения многих операций, например, таких как ввод/вывод, вычисление математических функций, обработка строк и т.п. Поэтому в большинстве программ присутствуют вызовы различных функций, хранящихся в *библиотеке стандартных функций C*.

Все компиляторы C поставляются вместе с библиотекой стандартных функций, предназначенных для выполнения наиболее общих задач. Стандарт C определяет минимальный набор функций, которые должны поддерживаться каждым компилятором. Но обычно библиотеки, поставляемые с компиляторами, имеют и много других, дополнительных, функций. Например, в стандартной библиотеке нет функций для работы с графикой, зато они есть почти в каждом компиляторе.

При вызове библиотечной функции компилятор “запоминает” ее имя. Потом компоновщик связывает код исходной программы с объектным кодом, уже найденным в стандартной библиотеке. Этот процесс называется *компоновкой*¹. У некоторых компиляторов есть свой собственный компоновщик, другие пользуются стандартным компоновщиком, поставляемым вместе с операционной системой.

В библиотеке функции хранятся в *переместимом* формате. Это значит, что адреса машинных инструкций в памяти являются не абсолютными, а относительными. При

¹ Или редактированием связей. — Прим. ред.

компоновке программы с функциями из стандартной библиотеки эти *относительные адреса*, или *смещения*, используются для определения действительных адресов. Для того чтобы научиться программировать на С (а значит и понять дальнейший материал данной книги), этого объяснения достаточно, более подробно процесс настройки адресов изложен в других книгах.

Библиотека стандартных функций содержит большое количество функций, необходимых для написания программы. Это своего рода кирпичики, из которых программист собирает программу. Кроме того, программист может написать свою функцию и поместить ее в библиотеку.

Раздельная компиляция

Короткая программа на языке С может состоять всего лишь из одного файла исходного текста. Однако при увеличении длины программы увеличивается также и время компиляции (при этом чем длиннее компиляция, тем короче терпение программиста!). Программа на С может состоять из двух или более файлов, компилируемых отдельно. Скомпилированные файлы программы komponуются с процедурами из библиотеки, формируя таким образом объектный код программы. Преимущество раздельной компиляции состоит в том, что при изменении одного файла нет необходимости перекомпилировать заново всю программу. При работе со сложными проектами это экономит много времени. Раздельная компиляция позволяет также нескольким программистам работать над одним проектом, так как она служит средством организации исходного текста программы для большого проекта. Более детально использование средств раздельной компиляции изложено в части V данной книги.

Компиляция программы на языке С

Создание выполнимой программы на языке С состоит из следующих трех шагов: разработка, компиляция и компоновка программы с библиотечными функциями.

В настоящее время большинство компиляторов поставляется вместе с оболочкой программирования, содержащей редактор текста. Оболочки содержат обычно также автономный компилятор. При наличии автономного компилятора для написания программы можно использовать любой удобный редактор. В противном случае нужно быть очень внимательным, так как встроенный компилятор нормально работает только со стандартным текстовым файлом. Например, компиляторы не могут обрабатывать файлы, созданные некоторыми текстовыми процессорами, так как эти файлы содержат управляющие коды и непечатаемые символы.

Конкретный способ компиляции программы зависит от типа используемого компилятора. Для разных компиляторов и оболочек способы компоновки также могут быть разными, например, компоновка может выполняться компилятором, а может и отдельной программой. Эти вопросы обычно освещаются в документации компилятора.

Карта памяти программы на языке С

Скомпилированная программа С имеет четыре логически обособленные области памяти. Первая — это область памяти, содержащая выполнимый код программы. Во второй области хранятся глобальные переменные. Оставшиеся две области — это стек

и динамически распределяемая область памяти¹. Стек используется для хранения вспомогательных переменных во время выполнения программы. Здесь находятся адреса возврата функций, аргументы функций, локальные переменные и т.п. Текущее состояние процессора также хранится в стеке. Динамически распределяемая область памяти, или куча — это такая свободная область памяти, для получения участков памяти из которой программа вызывает функции динамического распределения памяти.

На рис. 1.2 показано, как распределяется память во время выполнения программы. Но не следует забывать, что конкретное распределение может быть разным в зависимости от типа процессора и реализации языка.

■ Сравнительная характеристика языков С и С++

В заключение необходимо сказать несколько слов о языке С++. Начинающие программисты не всегда ясно представляют, что такое С++ и чем именно он отличается от С. В нескольких словах, язык С++ — это объектно-ориентированный язык программирования, фундаментом которого является С. Язык С — это подмножество С++ и, следовательно, С++ — надмножество С.



Рис. 1.2. Распределение памяти (карта памяти) при выполнении программы, написанной на языке С

В общем случае компилятор С++ можно использовать для компиляции программы, написанной на С. В настоящее время большинство компиляторов могут работать с программами, написанными как на С, так и на С++. Поэтому многие программисты используют компилятор С++ для компиляции программы, написанной на С. Но, поскольку С++ основан на стандарте С89, при написании программы С, рассчитан-

¹ Называется также динамической областью, динамически распределяемой областью, кучей и неупорядоченным массивом. — Прим. ред.

ной на компилятор C++, допускается использование только тех возможностей языка, которые предусмотрены в C89 (они рассматриваются в части I).

При написании программы на C, рассчитанной на компилятор C++, необходимо правильно указывать расширение файла, содержащего текст программы. Согласно действующему соглашению, файлы программ, написанных на C имеют расширение .C, а написанных на C++ — .CPP. Присвоение расширения .CPP файлу программы, написанной на C, недопустимо, потому как эти языки все же существенно отличаются друг от друга, и компилировать программу на C так, будто это программа на C++, нельзя. Расширение .C указывает транслятору на то, что он должен компилировать программу, написанную именно на C.

На заметку

Полное описание языка C++ приведено в книге Herbert Schildt. C++: The Complete Reference (Berkeley, CA: Osborne/McGraw-Hill).

Словарь терминов

- **Исходный текст (или код) программы.** Текст программы, который можно прочесть. Обычно его и называют программой. Исходный текст программы вводится в компилятор C.
- **Объектный код.** Результат трансляции исходного текста в машинный код, который может быть прочитан и выполнен компьютером. Объектный код обычно вводится в компоновщик.
- **Компоновщик или редактор связей.** Программа, которая компоует (связывает) отдельно оттранслированные модули в одну программу. Компоновщик также присоединяет функции стандартной библиотеки C и функции, написанные программистом. Результатом работы компоновщика является выполняемая программа.
- **Библиотека.** Файл, содержащий стандартные функции, используемые программой. Этот файл содержит операции ввода/вывода и другие полезные функции.
- **Время компиляции.** Время, затраченное компьютером на компиляцию программы.
- **Время выполнения.** Время, затраченное компьютером на выполнение программы.

Полный
справочник по



Глава 2

Выражения

В этой главе рассматриваются выражения — фундаментальные элементы языка С. По сравнению с другими языками программирования выражения языка С гораздо более гибкие и мощные. Составляющими элементами выражения являются *данные* и *операторы*¹. Данные могут быть представлены переменными, константами или значениями, возвращаемыми функциями. В языке С есть различные типы данных и большое количество операторов.

Базовые типы данных

Стандарт С89 определяет пять фундаментальных типов данных: `char` — символьные данные, `int` — целые, `float` — с плавающей точкой, `double` — двойной точности, `void` — без значения. На основе этих типов формируются другие типы данных. Размер (объем занимаемой памяти) и диапазон значений этих типов данных для разных процессоров и компиляторов могут быть разными. Однако объект типа `char` всегда занимает 1 байт. Размер объекта `int` обычно совпадает с размером слова в конкретной среде программирования. В большинстве случаев в 16-разрядной среде (DOS или Windows 4.1) `int` занимает 16 битов, а в 32-разрядной (Windows 95/98/NT/2000) — 32 бита. Однако полностью полагаться на это нельзя, особенно при переносе программы в другую среду. Необходимо помнить, что стандарт С обуславливает только *минимальный диапазон значений* каждого типа данных, но не размер в байтах.

На заметку

Кроме перечисленных пяти типов, стандарт С99 определяет еще три: `_Bool`, `_Complex` и `_Imaginary`. Они описаны в части II.

Конкретный формат числа с плавающей точкой зависит от его реализации в трансляторе. Переменные типа `char` обычно используются для обозначения набора символов стандарта ASCII, символы, не входящие в этот набор, разными компиляторами обрабатываются по-разному.

Диапазон значений типов `float` и `double` зависит от формата представления чисел с плавающей точкой. Стандарт С определяет для чисел с плавающей точкой минимальный диапазон значений от $1E-37$ до $1E+37$. Минимальное количество цифр мантиссы для типов с плавающей точкой приведено в табл. 2.1.

Тип `void` служит для объявления функции, не возвращающей значения, или для создания универсального указателя. Оба эти применения будут рассмотрены в последующих главах.

¹ В оригинале *operators*. Вообще в контексте языков программирования английскому слову *operator* соответствует два понятия: *оператор* и *операция/знак операции*. (В данном случае автор имеет в виду, конечно, *знаки операций*.) Еще каких-нибудь 30-40 лет назад более предпочтительным термином в данном случае был бы *знак операции*, поскольку в языках программирования под оператором понимались более сложные конструкции. Вместе с тем в математике, особенно в векторном анализе, а значит и в теории дифференциальных уравнений и математической физике под оператором часто имели в виду именно операцию или ее знак. (Например, говорили: “применим оператор набла” и тут же навешивали знак ∇ !) Таким образом, использование термина *оператор* как синонима термина *операция/знак операции* имеет глубокие корни (ведь первые ЭВМ были сконструированы для решения задач математической физики) и очень давнюю (столетнюю!) традицию. Что же касается русскоязычной компьютерной литературы, то здесь тоже достаточно давно (хотя и не сто лет, конечно!) имеет место тенденция все более частого употребления термина *оператор* вместо термина *операция/знак операции*. Первой книгой на русском языке, посвященной языку С, был перевод легендарного учебника *The C Programming Language*, написанного Б. Керниганом и Д. Ритчи. В первом издании перевода использовался термин *операция*. Однако уже во втором издании (1992 г.) его заменил термин *оператор*! Конечно, такое употребление этого термина идет вразрез со “школьной” традицией, но, как показала многолетняя практика, каких-либо серьезных трудностей при этом не возникает. — *Прим. ред.*

Модификация базовых типов

Базовые типы данных (кроме `void`) могут иметь различные *спецификаторы*¹, предшествующие им в тексте программы. Спецификатор типа так изменяет значение базового типа, чтобы он более точно соответствовал своему назначению в программе. Полный список спецификаторов типов:

`signed`
`unsigned`
`long`
`short`

Базовый тип `int` может быть модифицирован каждым из этих спецификаторов. Тип `char` модифицируется с помощью `unsigned` и `signed`, `double` — с помощью `long`. (Стандарт C99 также позволяет модифицировать `long` с помощью `long`, создавая таким образом `long long`, см. часть II). В табл. 2.1 приведены все допустимые комбинации типов данных с их минимальным диапазоном значений и типичным размером. Обратите внимание, в таблице приведены *минимально возможные*, а не типичные диапазоны значений. Например, если в компьютере арифметические операции выполняются над числами в дополнительных кодах (а именно так спроектированы почти все компьютеры!), то в диапазон значений целых попадут все целые числа от -32767 до 32768 .

Таблица 2.1. Все типы данных, определенные Стандартом C

Тип	Типичный размер в битах	Минимально допустимый диапазон значений
<code>char</code>	8	от -127 до 127
<code>unsigned char</code>	8	от 0 до 255
<code>signed char</code>	8	от -127 до 127
<code>int</code>	16 или 32	от -32767 до 32767
<code>unsigned int</code>	16 или 32	от 0 до 65535
<code>signed int</code>	16 или 32	то же, что <code>int</code>
<code>short int</code>	16	от -32767 до 32767
<code>unsigned short int</code>	16	от 0 до 65535
<code>signed short int</code>	16	то же, что <code>short int</code>
<code>long int</code>	32	от $-2\,147\,483\,647$ до $2\,147\,483\,647$
<code>long long int</code>	64	от $-(2^{63}-1)$ до $(2^{63}-1)$, добавлен стандартом C99
<code>signed long int</code>	32	то же, что <code>long int</code>
<code>unsigned long int</code>	32	от 0 до $4\,294\,967\,295$
<code>unsigned long long int</code>	64	от 0 до $(2^{64}-1)$, добавлен в C99
<code>float</code>	32	от $1E-37$ до $1E+37$, с точностью не менее 6 значащих десятичных цифр
<code>double</code>	64	от $1E-37$ до $1E+37$, с точностью не менее 10 значащих десятичных цифр
<code>long double</code>	80	от $1E-37$ до $1E+37$, с точностью не менее 10 значащих десятичных цифр

¹ Называются также *описателями*, *модификаторами* и *квалификаторами*. — Прим. ред.

Для целых можно использовать спецификатор `signed`, но в этом нет необходимости, потому что при объявлении целого он предполагается по умолчанию. Спецификатор `signed` чаще всего используется для типа `char`, который в некоторых реализациях по умолчанию может быть беззнаковым.

Целые числа со знаком и без знака отличаются интерпретацией нулевого бита числа. Если целое объявлено со знаком, компилятор считает, что нулевой бит содержит знак числа. Если в нулевом бите записан 0, число считается положительным, а если 1 — отрицательным.

В большинстве реализаций отрицательные числа представлены в *двоичном дополнителном коде*. Это значит, что для отрицательного числа все биты, кроме нулевого, инвертируются, к полученному числу добавляется 1, а нулевой бит устанавливается в 1.

Целые числа со знаком используются почти во всех алгоритмах, но абсолютная величина наибольшего из них составляет примерно только половину максимального целого без знака. Например, знаковое целое число 32767 в двоичном коде имеет вид

```
01111111 11111111
```

Если в нулевой бит записать 1, то оно будет интерпретироваться как -1 . Однако если полученную запись рассматривать как представление числа, объявленного как `unsigned int`, то оно будет интерпретироваться как 65535.

Если спецификатор типа записать сам по себе (без следующего за ним базового типа), то предполагается, что он модифицирует тип `int`. Таким образом, следующие спецификаторы типов эквивалентны:

Спецификатор	То же самое
<code>signed</code>	<code>signed int</code>
<code>unsigned</code>	<code>unsigned int</code>
<code>long</code>	<code>long int</code>
<code>short</code>	<code>short int</code>

Хотя базовый тип `int` и предполагается по умолчанию, его, тем не менее, обычно указывают явно.

Имена переменных

В языке C имена переменных, функций, меток и т.п. называются *идентификаторами*. Длина идентификатора (количество символов, из которых состоит идентификатор) является натуральным числом, обычно идентификатор представляет собой последовательность из одного или нескольких символов. Первый символ должен быть буквой или символом подчеркивания, последующие символы должны быть буквами, цифрами или символами подчеркивания. Ниже приведены примеры правильных и неправильных записей идентификаторов:

Правильные	Неправильные
<code>count</code>	<code>1count</code>
<code>test23</code>	<code>hi!here</code>
<code>high_balance</code>	<code>high...balance</code>

В языке C длина идентификатора может быть любой, однако не все его символы будут значащими. Объясним это на примере внешних и внутренних идентификаторов. Внешние идентификаторы участвуют во внешнем процессе *компоновки*¹. Эти идентификаторы, называемые *внешними именами*, обозначают имена функций и глобальных

¹ Редактирование связей, или разрешение внешних ссылок. — Прим. ред.

переменных, которые используются совместно в различных исходных файлах. Если идентификатор не участвует в процессе редактирования внешних ссылок, то он называется *внутренним именем*. К таким именам принадлежат, например, имена локальных переменных. В стандарте C89 значащими являются как минимум первые 6 символов внешнего имени и первые 31 символ внутреннего имени. Стандарт C99 увеличил этот диапазон, в нем значащими являются для внешнего идентификатора первые 31 символ, а для внутреннего — первые 63 символа. Кстати, в C++ значащими являются как минимум 1024 символа любого идентификатора. Эти отличия необходимо учитывать при конвертировании программ, написанных на языках C89, C99 или просто C, в программы на C++.

Верхние и нижние регистры символов рассматриваются как различные. Следовательно, `count`, `Count` и `COUNT` — это три разных идентификатора.

Идентификатор не может совпадать с ключевым словом C или с именем библиотечной функции.

Переменные

Переменная — это именованный участок памяти, в котором хранится значение, которое может быть изменено программой. Все переменные перед их использованием должны быть объявлены. Общая форма *объявления*¹ имеет такой вид:

тип список_переменных;

Здесь *тип* означает один из базовых или объявленных программистом типов (если необходимо — с одним или несколькими спецификаторами), а *список_переменных* состоит из одного или более идентификаторов, разделенных запятыми. Ниже приведены примеры объявлений:

```
int i, j, l;  
short int si;  
unsigned int ui;  
double balance, profit, loss;
```

Необходимо помнить, что в C имя переменной никогда не определяет ее тип.

Где объявляются переменные

Объявление переменных может быть расположено в трех местах: внутри функции, в определении параметров функции и вне всех функций. Это — места объявлений соответственно локальных переменных, формальных параметров функций и глобальных переменных.

Локальные переменные

Переменные, объявленные внутри функций, называются *локальными переменными*. В некоторых книгах по C они называются *динамическими переменными*². В этой книге используется более распространенный термин *локальная переменная*. Локальную переменную можно использовать только внутри блока, в котором она объявлена. Иными словами, локальная переменная невидима за пределами своего блока. (Блок программы — это описания и инструкции, объединенные в одну конструкцию путем заключения их в фигурные скобки.)

¹ Называется также *описанием* или *декларацией*. — Прим. ред.

² А в книгах по C++ *переменной автоматического класса* памяти (т.е. такой, что создается при входе в блок, где она объявлена, и уничтожается при выходе из него). — Прим. ред.

Локальные переменные существуют только во время выполнения программного блока, в котором они объявлены, создаются они при входе в блок, а разрушаются — при выходе из него. Более того, переменная, объявленная в одном блоке, не имеет никакого отношения к переменной с тем же именем, объявленной в другом блоке.

Чаще всего блоком программы, в котором объявлены локальные переменные, является функция. Рассмотрим, например, следующие две функции:

```
void func1(void)
{
    int x;
    x = 10;
}

void func2(void)
{
    int x;
    x = -199;
}
```

Целая переменная *x* объявлена дважды: один раз в `func1()` и второй — в `func2()`. При этом переменная *x* в одной функции никак не связана и никак не влияет на переменную с тем же именем в другой функции. Это происходит потому, что локальная переменная видима только внутри блока, в котором она объявлена, за пределами этого блока она невидима.

В языке С есть ключевое слово `auto` (спецификатор класса памяти), которое можно использовать в объявлении локальной переменной. Однако так как по умолчанию предполагается, что все переменные, не являющиеся глобальными, являются динамическими, то ключевое слово `auto` почти никогда не используется, а поэтому в примерах в данной книге отсутствует.

Из соображений удобства и в силу устоявшейся традиции все локальные переменные функции чаще всего объявляются в самом начале функции, сразу после открывающейся фигурной скобки. Однако можно объявить локальную переменную и внутри блока программы (блок функции — это частный случай блока программы). Например:

```
void f(void)
{
    int t;

    scanf("%d%c", &t);

    if(t==1) {
        char s[80]; /* эта переменная создается только
                     при входе в этот блок */
        printf("Введите имя:");
        gets(s);
        /* некоторые операторы */
    }

    /* здесь переменная s невидима */
}
```

В этом примере локальная переменная *s* создается при входе в блок `if` и разрушается при выходе из него. Следовательно, переменная *s* видима только внутри блока `if` и не может быть использована ни в каких других местах, даже если они находятся внутри функции, содержащей этот блок.

Объявление переменных внутри блока программы помогает избежать нежелательных побочных эффектов. Переменная не существует вне блока, в котором она объявлена, следовательно, “посторонний” участок программы не сможет случайно изменить ее значение.

Если имена переменных, объявленных во внутреннем и внешнем (по отношению к нему) блоках совпадают, то переменная внутреннего блока “прячет” (т.е. скрывает, делает невидимой) переменную внешнего блока. Рассмотрим следующий пример:

```
#include <stdio.h>

int main(void)
{
    int x;

    x = 10;

    if(x == 10){
        int x; /* эта x прячет внешнюю x */

        x = 99;
        printf("Внутренняя x: %d\n", x);
    }

    printf("Внешняя x: %d\n", x);

    return 0;
}
```

Результат выполнения программы следующий:

```
Внутренняя x: 99
Внешняя x: 10
```

В этом примере переменная *x*, объявленная внутри блока *if*, делает невидимой внешнюю переменную *x*. Следовательно, внутренняя и внешняя *x* — это два разных объекта. Когда блок заканчивается, внешняя *x* опять становится видимой.

В стандарте C89 все локальные переменные должны быть объявлены в начале блока, до любого выполнимого оператора. Например, следующая функция вызовет ошибку компиляции в C89:

```
/* Эта функция вызывает ошибку компиляции
   на компиляторе C89
*/
void f(void)

    int i;

    i = 10;

    int j; /* Ошибка в этой строке */

    j = 20;
}
```

Однако в C99 (и в C++) эта функция вполне работоспособна, потому что в них локальная переменная может быть объявлена в любом месте внутри блока до ее первого использования.

Так как локальные переменные создаются и уничтожаются при каждом входе и выходе из блока, их значение теряется каждый раз, когда программа выходит из блока. Это необходимо учитывать при вызове функции. Локальная переменная создается

при входе в функцию и разрушается при выходе из нее. Это значит, что локальная переменная не сохраняет свое значение в период между вызовами (однако можно дать указание компилятору сохранить значение локальной переменной, для этого нужно объявить ее с модификатором `static`).

По умолчанию локальные переменные хранятся в стеке. Стек — динамически изменяющаяся область памяти. Вот почему в общем случае локальные переменные не сохраняют свое значение в период между вызовами функций.

Локальные переменные можно инициализировать каким-либо заранее заданным значением. Это значение будет присвоено переменной каждый раз при входе в тот блок программы, в котором она объявлена. Например, следующая программа напечатает число 10 десять раз:

```
#include <stdio.h>

void f(void);

int main(void)
{
    int i;
    for(i=0; i<10; i++) f();
    return 0;
}

void f(void)
{
    int j = 10;
    printf("%d ",j);
    j++; /* этот оператор не влияет на результат */
}
```

Формальные параметры функции

Если функция имеет аргументы, значит должны быть объявлены переменные, которые примут их значения. Эти переменные называются *формальными параметрами* функции. Внутри функции они фигурируют как обычные локальные переменные. Как показано в следующем фрагменте программы, они объявляются после имени функции внутри круглых скобок:

```
/* Возвращает 1, если в строке s содержится символ c, в противном
случае возвращает 0 */
int is_in(char *s, char c)
{
    while(*s)
        if(*s==c) return 1;
        else s++;

    return 0;
}
```

Функция `is_in()` имеет два параметра: `s` и `c`, она возвращает 1, если символ, записанный в переменной `s`, входит в строку `s`, в противном случае она возвращает 0.

Внутри функции формальные параметры ничем не отличаются от обычных локальных переменных, единственное их отличие состоит в том, что при входе в функцию они получают значения аргументов. Можно, например, присваивать параметру какое-либо значение или использовать его в выражении. Необходимо помнить, что, как и локальные переменные, формальные параметры тоже являются динамическими переменными и, следовательно, разрушаются при выходе из функции.

Глобальные переменные

В отличие от локальных, *глобальные переменные* видимы и могут использоваться в любом месте программы. Они сохраняют свое значение на протяжении всей работы программы. Чтобы создать глобальную переменную, ее необходимо объявить за пределами функции. Глобальная переменная может быть использована в любом выражении, независимо от того, в каком блоке это выражение используется.

В следующем примере переменная `count` объявлена вне каких бы то ни было функций. Ее объявление расположено перед `main()`, однако, оно может находиться в любом месте перед первым использованием этой переменной, но только не внутри функции. Объявлять глобальные переменные рекомендуется в верхней части программы.

```
#include <stdio.h>
int count; /* глобальная переменная count */

void func1(void);
void func2(void);

int main(void)
{
    count = 100;
    func1();

    return 0;
}

void func1(void)
{
    int temp;

    temp = count;
    func2();
    printf("count равно %d", count); /* напечатает 100 */
}

void func2(void)
{
    int count;

    for(count=1; count<10; count++)
        putchar('.');
}
```

Внимательно посмотрите на эту программу. Обратите внимание на то, что ни в `func1()`, ни в `func2()` нет объявления переменной `count`, однако они обе могут ее использовать. В `func2()` эта возможность не реализуется, так как в ней объявлена локальная переменная с тем же именем. Когда внутри `func2()` происходит обращение к переменной `count`, то это будет обращение к локальной, а не глобальной переменной. Таким образом, выполняется следующее правило: если локальная и глобальная переменные имеют одно и то же имя, то при обращении к ней внутри блока, в котором объявлена локальная переменная, происходит ссылка на локальную переменную, а на глобальную переменную это никак не влияет.

Глобальные переменные хранятся в отдельной фиксированной области памяти, созданной компилятором специально для этого. Глобальные переменные используются в тех случаях, когда разные функции программы используют одни и те же данные. Однако рекомендуется избегать излишнего использования глобальных

переменных, потому что они занимают память в течение всего времени выполнения программы, а не только тогда, когда они необходимы. Кроме того, и это еще более важно, использование глобальной переменной делает функцию менее универсальной, потому что в этом случае функция использует нечто, определенное вне ее. К тому же большое количество глобальных переменных легко приводит к ошибкам в программе из-за нежелательных побочных эффектов. При увеличении размера программы серьезной проблемой становится случайное изменение значения переменной где-то в другой части программы, а когда глобальных переменных много, предотвратить это очень трудно.

■ Четыре типа областей видимости

В предыдущем (как, впрочем, и в последующем) изложении для объяснения различий между идентификаторами, объявленными вне блока и внутри его, используются термины *глобальная переменная* и *локальная переменная*. Однако в языке С предусмотрено более тонкое подразделение этих двух широких категорий. Стандарт С определяет четыре типа *областей видимости*¹ идентификаторов:

Тип области видимости	Область видимости
область действия — файл (имя, объявленное вне всех блоков и классов, можно использовать в транслируемом файле, содержащем это объявление; такие имена называются глобальными (global))	Начинается в начале файла (<i>единица трансляции</i>) и кончается в конце файла. Такую область видимости имеют только идентификаторы, объявленные вне функций. Эти идентификаторы видимы в любом месте файла. Переменные с этой областью видимости являются глобальными
область действия — блок	Начинается открывающейся фигурной скобкой { блока и кончается с его закрытием скобкой }. Эту область видимости имеют также параметры функции. Переменные, имеющие такую область видимости, являются локальными в своем блоке
область действия — прототип функции	Идентификаторы, объявленные в прототипе функции, видимы внутри прототипа
область действия — функция (имена, объявленные в функции, могут быть использованы только в теле функции)	Начинается открывающейся фигурной скобкой { функции и кончается с ее закрытием скобкой }. Такую область видимости имеют только метки. Метка используется оператором goto и должна находиться внутри той же функции

В этой книге используется главным образом более общее подразделение на глобальные и локальные имена. Однако при необходимости более тонкого подразделения используются изложенные выше типы областей видимости.

■ Квалификаторы типа

В языке С определяются *квалификаторы типа*², указывающие на доступность и модифицируемость переменной. Стандарт C89 определяет два квалификатора: `const` и `volatile`. (C99 добавляет третий, `restrict`, описанный в части II.) Квалификатор типа должен предшествовать имени типа, который он квалифицирует (уточняет).

¹ Область видимости называется также контекстом или областью действия (имен). — Прим. ред.

² Называются также классификаторами, описателями, спецификаторами. — Прим. ред.

Квалификатор `const`

Переменная, к которой в объявлении (декларации) применен квалификатор `const`, не может изменять свое значение¹. Ее можно только инициализировать, то есть присвоить ей значение в начале выполнения программы. Компилятор может поместить переменную этого типа в постоянное запоминающее устройство, так называемое ПЗУ (ROM, read-only memory). Например, в объявлении

```
const int a=10;
```

создается переменная с именем `a`, причем ей присваивается начальное значение 10, которое в дальнейшем в программе изменить никак нельзя. Переменную, к которой в объявлении применен квалификатор `const`, можно использовать в различных выражениях. Однако свое значение она может получить только в результате инициализации или с помощью аппаратно-зависимых средств.

Квалификатор `const` часто используется для того, чтобы предотвратить изменение функцией объекта, на который указывает аргумент функции. Без него при передаче в функцию указателя эта функция может изменить объект, на который он указывает. Однако если в объявлении параметра-указателя применен квалификатор `const`, функция не сможет изменить этот объект. В следующем примере функция `sp_to_dash()` печатает минус вместо каждого пробела в строке, передаваемой ей как аргумент. То есть строка “тестовый пример” будет напечатана как “тестовый-пример”. Применение квалификатора `const` в объявлении параметра функции гарантирует, что внутри функции объект, на который указывает параметр функции, не будет изменен.

```
#include <stdio.h>

void sp_to_dash(const char *str);

int main(void)
{
    sp_to_dash("тестовый пример");

    return 0;
}

void sp_to_dash(const char *str)
{
    while(*str){
        if(*str== ' ') printf("%c", '-');
        else printf("%c", *str);
        str++;
    }
}
```

Если написать `sp_to_dash()` таким образом, что внутри функции строка изменится, то еще на этапе компиляции в программе будет обнаружена ошибка. Например, на этапе компиляции возникнет ошибка, если написать так:

```
/* Неправильный пример. */
void sp_to_dash(const char *str)
{
```

¹ На жаргоне программистов: переменная “типа” `const` не может изменять значение. В описании многих языков такие переменные часто называются константами. Но если исключить левые части операторов присваивания, то переменные этого “типа” могут использоваться в тех же ситуациях, что и “настоящие” переменные. В этом смысле константы являются частным случаем переменных. — *Прим. ред.*

```

while(*str){
    if(*str==' ') *str= '-'; /* Это неправильно */
    printf("%c", *str);
    str++;
}

```

Квалификатор `const` используется в объявлениях параметров многих функций стандартной библиотеки. Например, прототип функции `strlen()` выглядит так:

```
size_t strlen(const char *str);
```

Применение квалификатора `const` в объявлении `str` гарантирует, что функция не изменит строку, на которую указывает `str`. Если функция стандартной библиотеки не предназначена для изменения аргумента, то практически всегда в объявлении указателя на аргумент применяется квалификатор `const`.

Программист тоже может применять квалификатор `const` для того, чтобы гарантировать сохранность объекта. Но следует помнить, что переменная, даже если к ней применен квалификатор `const`, может быть изменена в результате какого-нибудь внешнего по отношению к программе воздействия. Например, ей может быть присвоено значение каким-либо устройством. Однако применение квалификатора `const` в объявлении переменной гарантирует, что ее изменение может произойти только в ходе внешнего по отношению к программе события.

Квалификатор `volatile`

Квалификатор `volatile` указывает компилятору на то, что значение переменной может измениться независимо от программы, т.е. вследствие воздействия еще чего-либо, не являющегося оператором программы. Например, адрес глобальной переменной можно передать в подпрограмму операционной системы, следящей за временем, и тогда эта переменная будет содержать системное время. В этом случае значение переменной будет изменяться без участия какого-либо оператора программы. Знание таких подробностей важно потому, что большинство компиляторов с автоматически оптимизируют некоторые выражения, предполагая при этом неизменность переменной, если она не встречается в левой части оператора присваивания. В этом случае при очередной ссылке на переменную может использоваться ее предыдущее значение. Некоторые компиляторы изменяют порядок вычислений в выражениях, что может привести к ошибке, если в выражении присутствует переменная, вычисляемая вне программы. Квалификатор `volatile` предотвращает такие изменения программы.

Квалификаторы `const` и `volatile` могут применяться и совместно. Например, если `0x30` — адрес порта, значение в котором может задаваться только извне, то следующее объявление предотвратит всякую возможность нежелательных побочных эффектов:

```
const volatile char *port = (const volatile char *) 0x30;
```



Спецификаторы класса памяти

Стандарт C поддерживает четыре спецификатора класса памяти:

```

extern
static
register
auto

```

Эти спецификаторы сообщают компилятору, как он должен разместить соответствующие переменные в памяти. Общая форма объявления переменных при этом такова:

спецификатор_класса_памяти тип имя_переменной;

Спецификатор класса памяти в объявлении всегда должен стоять первым.

На заметку

Стандарты C89 и C99 из соображений удобства синтаксиса утверждают, что typedef — это спецификатор класса памяти. Однако typedef не является собственно спецификатором. Подробнее typedef рассматривается в книге далее.

Спецификатор extern

Прежде чем приступить к рассмотрению спецификатора extern, необходимо коротко остановиться на компоновке программы. В языке C при редактировании связей к переменной может применяться одно из трех связываний: внутреннее, внешнее или же не относящееся ни к одному из этих типов. (В последнем случае редактирование связей к ней не применяется.) В общем случае к именам функций и глобальных переменных применяется внешнее связывание. Это означает, что после компоновки они будут доступны во всех файлах, составляющих программу. К объектам, объявленным со спецификатором static и видимым на уровне файла, применяется внутреннее связывание, после компоновки они будут доступны только внутри файла, в котором они объявлены. К локальным переменным связывание не применяется и поэтому они доступны только внутри своих блоков.

Спецификатор extern указывает на то, что к объекту применяется внешнее связывание, именно поэтому они будут доступны во всей программе. Далее нам понадобятся чрезвычайно важные понятия объявления и описания. *Объявление (декларация)* объявляет имя и тип объекта. *Описание*¹ выделяет для объекта участок памяти, где он будет находиться. Один и тот же объект может быть объявлен неоднократно в разных местах, но описан он может быть только один раз.

В большинстве случаев объявление переменной является в то же время и ее описанием. Однако, если перед именем переменной стоит спецификатор extern, то объявление переменной может и не быть ее описанием. Таким образом, если нужно сослаться на переменную, определенную в другой части программы, необходимо объявить ее как внешнюю (extern).

Приведем пример использования спецификатора extern. Обратите внимание, что глобальные переменные first и last объявлены *после* main().

```
#include <stdio.h>

int main(void)
{
    extern int first, last;
    /* используются глобальные переменные */

    printf("%d %d", first, last);

    return 0;
}

/* описание глобальных переменных first и last */
int first = 10, last = 20;
```

¹ Синонимы: *определение, дефиниция*. — Прим. ред.

Программа напечатает 10 20, потому что глобальные переменные `first` и `last` инициализированы этими значениями. Объявление `extern` сообщает компилятору, что переменные `first` и `last` определены в другом месте, поэтому программа компилируется без ошибки, несмотря даже на то, что `first` и `last` используются до своего описания.

Обратите внимание, в этом примере объявление переменных со спецификатором `extern` необходимо только потому, что они не были объявлены до `main()`. Если бы их объявление встретилось перед `main()`, то в объявлении со спецификатором `extern` не было бы необходимости.

При компиляции выполняются следующие правила. Если компилятор находит переменную, не объявленную внутри блока, он ищет ее объявление во внешних блоках. Если не находит ее и там, то ищет среди объявлений глобальных переменных. В предыдущем примере, если бы не было объявления `extern`, компилятор не нашел бы `first` и `last` среди глобальных переменных, потому что они объявлены после `main()`. Здесь спецификатор `extern` сообщает компилятору, что эти переменные будут объявлены в файле позже.

Как сказано выше, спецификатор `extern` позволяет объявить переменную, не описывая ее. Но если в объявлении со спецификатором `extern` инициализировать переменную, то это объявление становится также и описанием. При этом программист обязательно должен учитывать, что объект может иметь много объявлений, но лишь одно описание.

Спецификатор `extern` играет большую роль в программах, состоящих из многих файлов. В языке C программа может быть записана в нескольких файлах, которые компилируются раздельно, а затем компонуются в одно целое. В этом случае необходимо как-то сообщить всем файлам о глобальных переменных программы. Самый лучший (и наиболее переносимый) способ сделать это — определить (описать) все глобальные переменные в одном файле и объявить их со спецификатором `extern` в остальных файлах, как показано на рис. 2.1.

Первый файл

```
int x, y;
char ch;
int main(void)
{
    /* ... */
}

void func1(void)
{
    x=123;
}
```

Второй файл

```
extern int x, y;
extern char ch;
void func22(void)
{
    x = y / 10;
}

void func23(void)
{
    y = 10;
}
```

Рис 2.1. Использование глобальных переменных в раздельно компилируемых модулях

Во втором файле спецификатор `extern` сообщает компилятору, что эти переменные определены в других файлах. Таким образом компилятор узнает имена и типы переменных, размещенных в другом месте, и может отдельно компилировать второй файл, ничего не зная о первом. При компоновке этих двух модулей все ссылки на глобальные переменные будут разрешены.

На практике программисты обычно включают объявления `extern` в заголовочные файлы, которые просто подключаются к каждому файлу исходного текста программы. Это более легкий путь, который к тому же приводит к меньшему количеству ошибок, чем повторение этих объявлений вручную в каждом файле.

Спецификатор static

Переменные, объявленные со спецификатором `static`, хранятся постоянно внутри своей функции или файла. В отличие от глобальных переменных они невидимы за пределами своей функции или файла, но они сохраняют свое значение между вызовами. Эта особенность делает их полезными в общих и библиотечных функциях, которые будут использоваться другими программистами. Спецификатор `static` воздействует на локальные и глобальные переменные по-разному.

Локальные статические переменные

Для локальной переменной, описанной со спецификатором `static`, компилятор выделяет в постоянное пользование участок памяти, точно так же, как и для глобальных переменных. Коренное отличие статических локальных от глобальных переменных заключается в том, что статические локальные переменные видны только внутри блока, в котором они объявлены. Говоря коротко, статические локальные переменные — это локальные переменные, сохраняющие свое значение между вызовами функции.

Статические локальные переменные очень важны при создании функций, работающих отдельно, так как многие процедуры требуют сохранения некоторых значений между вызовами. Если бы не было статических переменных, вместо них пришлось бы использовать глобальные, подвергая их риску непреднамеренного изменения другими участками программы. Рассмотрим пример функции, в которой особенно уместно применение статической локальной переменной. Это — генератор последовательности чисел, каждое из которых зависит только от предыдущего. Для хранения числа между вызовами можно использовать глобальную переменную. Однако тогда при каждом использовании функции придется объявлять эту переменную и, что особенно неудобно, постоянно следить за тем, чтобы ее объявление не конфликтовало с объявлениями других глобальных переменных. Значительно лучшее решение — объявить эту переменную со спецификатором `static`:

```
int series(void)
{
    static int series_num;

    series_num = series_num+23;
    return series_num;
}
```

В этом примере переменная `series_num` продолжает существовать между вызовами функций, в то время как обычная локальная переменная создается заново при каждом вызове, а затем уничтожается. Поэтому в данном примере каждый вызов `series()` генерирует новое число, зависящее от предыдущего, причем удается обойтись без глобальных переменных.

Статическую локальную переменную можно инициализировать. Это значение присваивается ей только один раз — в начале работы всей программы, но не при каждом входе в блок программы, как обычной локальной переменной. В следующей версии функции `series()` статическая локальная переменная инициализируется числом 100:

```
int series(void)
{
    static int series_num = 100;
```

```

series_num = series_num+23;
return series_num;
}

```

Теперь эта функция всегда будет генерировать последовательность, начинающуюся с числа 123. Однако во многих случаях необходимо дать пользователю программы возможность ввести первое число вручную. Для этого переменную `series_num` можно сделать глобальной и предусмотреть возможность задания начального значения. Если же отказаться от объявления переменной `series_num` в качестве глобальной, то необходимо ее объявить со спецификатором `static`.

Глобальные статические переменные

Спецификатор `static` в объявлении глобальной переменной заставляет компилятор создать глобальную переменную, видимую только в том файле, в котором она объявлена. Статическая глобальная переменная, таким образом, подвергается внутреннему связыванию, как описано ранее в разделе “Спецификатор `extern`”. Это значит, что хоть эта переменная и глобальная, тем не менее процедуры в других файлах не увидят ее и не смогут случайно изменить ее значение. Этим снижается риск нежелательных побочных эффектов. А в тех относительно редких случаях, когда для выполнения задачи статическая локальная переменная не подойдет, можно создать небольшой отдельный файл, который содержит только функции, в которых используется эта статическая глобальная переменная. Затем этот файл необходимо откомпилировать отдельно; тогда можно быть уверенным, что побочных эффектов не будет.

В следующем примере иллюстрируется применение статической глобальной переменной. Здесь генератор последовательности чисел переделан так, что начальное число задается вызовом другой функции, `series_start()`:

```

/* Это должно быть в одном файле
   отдельно от всего остального */

static int series_num;
void series_start(int seed);
int series(void);

int series(void)
{
    series_num = series_num+23;
    return series_num;
}

/* инициализирует переменную series_num */
void series_start(int seed)
{
    series_num = seed;
}

```

Вызов функции `series_start()` с некоторым целым числом в качестве параметра инициализирует генератор `series()`. После этого можно генерировать последовательность чисел путем многократного вызова `series()`.

Обзор: Имена локальных статических переменных видимы только внутри блока, в котором они объявлены; имена глобальных статических переменных видимы только внутри файла, в котором они объявлены.

Если поместить функции `series()` и `series_num()` в библиотеку, то уже нельзя будет сослаться на переменную `series_num`, она оказалась спрятанной от любых операторов всей остальной программы. При этом в программе (конечно, в других файлах) можно объявить и использовать другую переменную под именем `series_num`. Иными словами, спе-

цификатор `static` позволяет создать переменную, видимую только для функций, в которых она нужна, что исключает нежелательные побочные эффекты.

Таким образом, при разработке больших и сложных программ для “сокрытия” переменных можно применять спецификатор `static`.

Спецификатор `register`

Первоначально спецификатор класса памяти `register` применялся только к переменным типа `int`, `char` и для указателей. Однако стандарт C расширил использование спецификатора `register`, теперь он может применяться к переменным любых типов.

В первых версиях компиляторов C спецификатор `register` сообщал компилятору, что переменная должна храниться в регистре процессора, а не в оперативной памяти, как все остальные переменные. Это приводит к тому, что операции с переменной `register` осуществляются намного быстрее, чем с обычными переменными, потому такая переменная уже находится в процессоре и не нужно тратить время на выборку ее значения из оперативной памяти (и на запись в память).

В настоящее время определение спецификатора `register` существенно расширено. Стандарты C89 и C99 попросту декларируют “доступ к объекту так быстро, как только возможно”. Практически при этом символьные и целые переменные по-прежнему размещаются в регистрах процессора. Конечно, большие объекты (например, массивы) не могут поместиться в регистры процессора, однако компилятор получает указание “позаботиться” о быстродействии операций с ними. В зависимости от конкретной реализации компилятора и операционной системы переменные `register` обрабатываются по-разному. Иногда спецификатор `register` попросту игнорируется, а переменная обрабатывается как обычная, однако на практике это бывает редко.

Спецификатор `register` можно применить только к локальным переменным и формальным параметрам функций. В объявлении глобальных переменных применение спецификатора `register` не допускается. Ниже приведен пример использования переменной, в объявлении которой применен спецификатор `register`; эта переменная используется в функции возведения целого числа `m` в степень. (Степень — натуральное число — представлена идентификатором `e`.)

```
int int_pwr(register int m, register int e)
{
    register int temp;

    temp = 1;

    for(; e; e--) temp = temp * m;
    return temp;
}
```

В этом примере в объявлениях к переменным `e`, `m` и `temp` применен спецификатор `register` потому, что они используются внутри цикла. Переменные `register` идеально подходят для оптимизации скорости работы цикла. Как правило, переменные `register` используются там, где от них больше всего пользы, а именно, когда процесс многократно обращается к одной и той же переменной. Это существенно потому, что в объявлении можно применить спецификатор `register` к любой переменной, но средства оптимизации быстродействия могут быть применены далеко не ко всем переменным в равной степени.

Максимальное количество переменных `register`, оптимизирующихся по быстродействию, зависит от среды программирования и конкретной реализации компилятора. Если таких переменных окажется слишком много, то компилятор автоматически

преобразует регистровые переменные в нерегистровые. Этим обеспечивается переносимость программы в широком диапазоне процессоров.

Обычно в регистры процессора можно поместить как минимум две переменные типа `char` или `int`. Однако в различных средах программирования режимы оптимизации могут очень отличаться, поэтому выбор режима оптимизации необходимо осуществлять с учетом особенностей конкретного компилятора.

В языке С с помощью оператора `&` (рассматривается в этой главе далее) нельзя получить адрес регистровой переменной, потому что она может храниться в регистре процессора, который обычно не имеет адреса.

Хотя в настоящее время применение спецификатора `register` в значительной мере вышло за его традиционные рамки, практически ощутимый эффект от его применения по-прежнему может быть получен только для переменных целого и символьного типа. Не следует ожидать заметного повышения скорости от объявления регистровыми переменных других типов.



Инициализация переменных

При объявлении переменной она может быть инициализирована. Для этого нужно после ее объявления поставить знак равенства и константу, т.е. общая форма инициализации имеет следующий вид:

тип имя_переменной = константа;

Приведем несколько примеров инициализации переменных:

```
char ch = 'a';
int first = 0;
double balance = 123.23;
```

Глобальные и статические локальные переменные инициализируются только один раз в начале работы программы. А локальные переменные (исключая статические локальные) инициализируются каждый раз при входе в блок, в котором они объявлены. Неинициализированные локальные переменные до первого присвоения имеют произвольное значение. Неинициализированные глобальные и статические локальные переменные в начале работы программы автоматически обнуляются.



Константы

Константа — это фиксированное значение, которое не может быть изменено программой. Константа может относиться к любому базовому типу. Способ представления константы определяется ее типом. Константы также называются *литералами*.

Символьные константы заключаются в одинарные кавычки. Например, `'a'` и `'%'` — это символьные константы. В языке С определены многобайтовые (состоящие из одного или более байт) и широкие (обычно длиной 16 бит) символы. Они используются для представления символов языков, имеющих в своем алфавите много букв. Многобайтовый символ записывается в одинарных кавычках, например, `'ху'`, а широкий — с предшествующим символом `L`, например:

```
wchar_t wc;
wc = L'A';
```

Здесь переменной `wc` присвоено значение константы `A`, рассматриваемой как широкий символ. Тип широкого символа `wchar_t` определен в заголовочном файле `<stddef.h>`, этот тип не является встроенным.

Целые константы определяются как числа без дробной части. Например, 10 и –100 — это целые константы. Константы в плавающем формате записываются как числа с десятичной точкой, например, 11.123. Допускается также экспоненциальное представление чисел (в виде мантиссы и порядка): 111.23e–1.

По умолчанию компилятор приписывает константе тип наименьшего размера, в ячейку которого может уместиться константа. Таким образом, если целые числа обычно являются 16-разрядными, то константа 10 по умолчанию имеет тип `int`, а 103000 — тип `long int`. Число 10 может поместиться в типе `char`, однако компилятор не нарушит границы типов и поместит ее в `int`. Но это правило имеет исключение: всем константам в плавающем формате, даже самым маленьким, приписывается тип `double` (если, конечно, они сюда помещаются).

Определение типов констант по умолчанию является вполне удовлетворительным при разработке большинства программ. Однако, используя суффикс, можно явно указать тип числовой константы. Если после числа в плавающем формате стоит суффикс `F`, то считается, что константа имеет тип `float`, а если `L`, то `long double`. Для целых типов суффикс `U` означает `unsigned`, а `L` — `long`. Тип суффикса не зависит от регистра, например, как `F`, так и `f` определяют константы типа `float`. Приведем несколько примеров:

Тип данных	Примеры констант				
<code>int</code>	1	123	21000	–243	
<code>long int</code>	35000L	–34L			
<code>unsigned int</code>	10000U	987u	40000U		
<code>float</code>	123.23F	4.34e–4f			
<code>double</code>	123.23	1.0	–0.9876324		
<code>long double</code>	1001.2L				

Стандарт C99 определяет также целые константы типа `long long`, их суффикс — `LL` или `ll`.

Шестнадцатеричные и восьмеричные константы

Иногда удобнее использовать не десятичную, а восьмеричную или шестнадцатеричную систему. Позиционную систему счисления с основанием 8 называют *восьмеричной*. В ней используются цифры от 0 до 7. Число 10 в восьмеричной системе представляет то же число, что и 8 в десятичной. Позиционная система счисления с основанием 16 называется шестнадцатеричной. В ней используются 16 символов: цифры от 0 до 9 и символы от `A` до `F`, обозначающие цифры от 10 до 15. Например, запись 10 в шестнадцатеричной системе обозначает то же число, что и 16 в десятичной системе. Эти системы счисления используются довольно часто, поэтому в C целые константы можно определять не только в десятичной, но и в восьмеричной и шестнадцатеричной системах счисления. Шестнадцатеричная константа начинается с `0x`, а восьмеричная — с `0`, например:

```
| int hex = 0x80; /* 128 в десятичной системе */
| int oct = 012; /* 10 в десятичной системе */
```

Строковые константы

Язык C поддерживает еще один тип констант, а именно — строковые. *Строка* — это последовательность символов, заключенных в двойные кавычки. Например, "тест" — это строка. В этой книге ранее уже встречались примеры строк в функции `printf()`. В термине "строковая константа" слово "строковая" не означает строковый предопределенный тип данных, такого в C нет, здесь это всего лишь прилагательное.

Не следует путать понятия строки и символа. Символьная константа заключается в одинарные кавычки, например, 'а'. Соответственно запись "а" означает строку, состоящую из одного символа.

Специальные символьные константы

Чтобы представить большинство символьных констант, достаточно заключить соответствующий символ в одинарные кавычки. Но некоторые символы, например, символ возврата каретки, требуют специального представления. В языке С определены *специальные символьные константы*, приведенные в табл. 2.2. Иногда их называют ESC-последовательностями, управляющими последовательностями и символами с обратным слэшем. Управляющие последовательности можно использовать вместо ASCII-кодов для обеспечения лучшей переносимости программы.

В следующем примере программа выводит символ новой строки (т.е. переходит на новую строку), выводит символ табуляции (т.е. переходит на первую позицию табуляции) и, наконец, выводит строку Простой тест.

```
#include <stdio.h>

int main(void)
{
    printf("\n\tПростой тест.");

    return 0;
}
```

Таблица 2.2. Специальные символьные константы

Код	Назначение
\b	Удаление предыдущего символа
\f	Подача бумаги
\n	Новая строка
\r	Возврат каретки
\t	Горизонтальная табуляция
\"	Двойная кавычка
'	Одинарная кавычка
\\	Обратный слэш
\v	Вертикальная табуляция
\a	Сигнал
\?	Знак вопроса
\N	Восьмеричная константа (N — восьмеричное представление)
\xN	Шестнадцатеричная константа (N — шестнадцатеричное представление)

Операции

Язык С содержит большое количество встроенных операций. Их роль в С значительно больше, чем в других языках программирования. Существует четыре основных класса операций: *арифметические, логические, поразрядные и операции сравнения*. Кроме них, есть также некоторые специальные операторы, например, оператор присваивания.

Оператор присваивания

Оператор присваивания может присутствовать в любом выражении языка С¹. Этим С отличается от большинства других языков программирования (Pascal, BASIC и FORTRAN), в которых присваивание возможно только в отдельном операторе. Общая форма оператора присваивания:

имя_переменной=выражение;

Выражение может быть просто константой или сколь угодно сложным выражением. В отличие от Pascal или Modula-2, в которых для присваивания используется знак “:=”, в языке С оператором присваивания служит единственный знак присваивания “=”. *Адресатом (получателем)*, т.е. левой частью оператора присваивания должен быть объект, способный получить значение, например, переменная.

В книгах по С и в сообщениях компилятора часто встречаются термины *lvalue*² (*left side value*) и *rvalue*³ (*right side value*). Попросту говоря, *lvalue* — это объект. Если этот объект может стоять в левой части присваивания, то он называется также *модифицируемым (modifiable) lvalue*. Подытожим сказанное: *lvalue* — это объект в левой части оператора присваивания, получающий значение, чаще всего этим объектом является переменная. Термин *rvalue* означает значение выражения в правой части оператора присваивания.

Преобразование типов при присваиваниях

Если в операции встречаются переменные разных типов, происходит *преобразование типов*. В операторе присваивания действует простое правило: значение выражения в правой части преобразуется к типу объекта в левой части.

```
int x;
char ch;
float f;

void func(void)
{
    ch = x;      /* 1-я строка */
    x = f;      /* 2-я строка */
    f = ch;     /* 3-я строка */
    f = x;      /* 4-я строка */
}
```

В 1-й строке этого примера старшие двоичные разряды целой переменной *x* отбрасываются, а в *ch* заносятся младшие 8 бит. Если значение *x* лежит в интервале от 0 до 255, то *ch* и *x* будут идентичны и потери информации не произойдет. В противном случае в *ch* будут занесены только младшие разряды переменной *x*. Во 2-й строке в *x* будет записана целая часть числа *f*. В 3-й строке произойдет преобразование целого 8-разрядного числа, хранящегося в *ch*, в число в плавающем формате. В 4-й строке произойдет то же самое, только с 16-разрядным целым.

¹ В данном случае под оператором имеется в виду, конечно, знак операции. По этому поводу см. сделанное ранее примечание редактора о переводе термина *operator*. — Прим. ред.

² *lvalue* — именуемое выражение, т.е. выражение, которое может стоять в левой части оператора присваивания. Под *lvalue* также часто подразумевается адрес переменной. (С идентификатором переменной в программе связано две величины: адрес переменной и ее значение. Адрес используется, когда переменная стоит в левой части присваивания, значение — в правой части присваивания.) Иногда встречается и термин *l-значение*. Как бы то ни было, этим термином обозначается выражение, которое может находиться в левой части оператора присваивания. Семантически оно представляет собой адрес, по которому размещена переменная, массив, элемент структуры и т.п. — Прим. ред.

³ *rvalue* — значение переменной; иногда переводится, как *r-значение*, т.е. значение в правой части оператора присваивания. — Прим. ред.

Преобразование целых в символы и длинных целых в целые удаляет соответствующее количество старших двоичных разрядов. В 16-разрядной среде теряются 8 битов при преобразовании целого в символ и 16 битов при преобразовании длинного целого в целое. В 32-разрядной среде теряются 24 бита при преобразовании целого в символ и 16 битов при преобразовании целого в короткое целое.

В табл. 2.3. приведены варианты потери информации при некоторых преобразованиях. Необходимо помнить, что преобразование `int` во `float` или `float` в `double` не повышает точность вычислений. При таком преобразовании только изменяется форма представления числа. Некоторые компиляторы при преобразовании `char` в `int` считают переменную `char` положительной независимо от ее значения. Другие компиляторы считают переменную `char` отрицательной, если она больше 127. Поэтому для обеспечения переносимости программы необходимо использовать переменные типа `char` для хранения символов, а переменные типа `signed char` и `int` (целый) — для хранения чисел.

Таблица 2.3. Результат некоторых преобразований типов

Тип адресата	Тип выражения	Потеря информации
<code>signed char</code>	<code>char</code>	Если значение > 127, то результат отрицательный
<code>char</code>	<code>short int</code>	Старшие 8 бит
<code>char</code>	<code>int</code> (16- разрядный)	Старшие 8 бит
<code>char</code>	<code>int</code> (32- разрядный)	Старшие 24 бита
<code>char</code>	<code>long int</code>	Старшие 24 бита
<code>short int</code>	<code>int</code> (16- разрядный)	Нет
<code>short int</code>	<code>int</code> (32- разрядный)	Старшие 16 бит
<code>int</code> (16-разрядный)	<code>long int</code>	Старшие 16 бит
<code>int</code> (32- разрядный)	<code>long int</code>	Нет
<code>long int</code> (32- разрядный)	<code>long long int</code> (64- разрядный)	Старшие 32 бита (это относится только к C99)
<code>int</code>	<code>float</code>	Дробная часть
<code>float</code>	<code>double</code>	Результат округляется
<code>double</code>	<code>long double</code>	Результат округляется

Если какое-либо преобразование не приведено в табл. 2.3, то, чтобы определить, что именно теряется в результате этого преобразования, нужно представить его в виде композиции (суперпозиции, произведения) указанных в таблице преобразований и затем провести последовательные преобразования. Например, преобразование `double` в `int` эквивалентно последовательному выполнению двух преобразований: сначала `double` в `float`, а затем `float` в `int`.

Множественное присваивание

В одном операторе присваивания можно присвоить одно и то же значение многим переменным. Для этого используется оператор *множественного присваивания*¹, например:

```
x = y = z = 0;
```

Следует отметить, что в практике программирования этот прием используется очень часто.

¹ *Множественное присваивание* — присваивание одного и того же значения нескольким переменным. Под множественным присваиванием также подразумевается конструкция языка программирования, позволяющая присвоить одно и то же значение нескольким переменным одновременно. — *Прим. ред.*

Составное присваивание

Составное присваивание — это разновидность оператора присваивания, в которой запись сокращается и становится более удобной в написании¹. Например, оператор

```
x = x+10;
```

можно записать как

```
x += 10;
```

Оператор “+=” сообщает компилятору, что к переменной *x* нужно прибавить 10.

“Составные” операторы² присваивания существуют для всех бинарных операций (то есть операций, имеющих два операнда). Любой оператор вида

переменная = переменная оператор выражение;

можно записать как

переменная оператор= выражение;

Еще один пример:

```
x = x-100;
```

означает то же самое, что и

```
x -= 100;
```

Составное присваивание значительно компактнее, чем соответствующее простое присваивание, поэтому его иногда называют *стенографическим (shorthand) присваиванием*. В программах на С этот оператор широко используется, поэтому необходимо хорошо его усвоить.

Арифметические операции

В табл. 2.4 приведены арифметические операции С. Операции +, −, * и / работают так же, как и в большинстве других языков программирования. Их можно применять почти ко всем встроенным типам данных. Если операция / применяется к целому или символьному типам, то остаток от деления отбрасывается. Например, результатом операции 5/2 является 2.

Оператор деления по модулю % в С работает так же, как и в других языках, его результатом является остаток от целочисленного деления. Этот оператор, однако, нельзя применять к типам данных с плавающей точкой. Применение оператора % иллюстрируется следующим примером:

```
int x, y;

x = 5;
y = 2;

printf("%d ", x/y); /* напечатает 2 */
printf("%d ", x%y); /* напечатает 1,
                        остаток от целочисленного деления */
```

¹ По этой причине варианты оператора присваивания, в которых используется такая запись, называются “сокращенными” или “укороченными”. Что касается терминологии, то необходимо отметить также следующее обстоятельство. Хотя термины *присваивание* и *оператор присваивания* часто могут рассматриваться как синонимы, составное присваивание не является составным оператором! (Под составным оператором в языке С подразумевают блок.) — *Прим. ред.*

² Под “составными” операторами в данном случае, конечно, подразумеваются составные знаки операций, т.е. знаки операций, состоящие из нескольких (обычно двух) символов. Составные операторы-блоки не имеют к этому никакого отношения. — *Прим. ред.*

```
x = 1;
y = 2;

printf("%d %d ", x/y, x*y); /* напечатает 0 1 */
```

Последняя строка программы напечатает 0 1 потому, что при целочисленном делении остаток отбрасывается и здесь результат будет 0, а сам остаток равен 1.

Таблица 2.4. Арифметические операции

Оператор	Операция
-	Вычитание, также унарный минус
+	Сложение
*	Умножение
/	Деление
%	Остаток от деления
--	Декремент ¹ , или уменьшение
++	Инкремент ² , или увеличение

Унарный минус умножает операнд на -1 , то есть меняет его знак на противоположный.

Операции увеличения (инкремента) и уменьшения (декремента)

В языке C есть два полезных оператора, значительно упрощающие широко распространенные операции. Это инкремент ++ и декремент --. Оператор ++ увеличивает значение операнда на 1, а -- уменьшает на 1. Иными словами,

```
x = x+1;
```

можно записать как

```
++x;
```

Аналогично оператор

```
x = x-1;
```

равносилен оператору

```
x--;
```

Как инкремент, так и декремент могут предшествовать операнду (префиксная форма) или следовать за ним (постфиксная форма). Например

```
x = x+1;
```

можно записать как в виде

```
++x;
```

так и в виде

```
x++;
```

Однако префиксная и постфиксная формы отличаются при использовании их в выражениях. Если оператор инкремента или декремента предшествует операнду, то сама операция выполняется до использования результата в выражении. Если же оператор следует за операндом, то в выражении значение операнда используется до вы-

¹ На жаргоне программистов: *декрементация*. — Прим. ред.

² На жаргоне программистов: *инкрементация*. — Прим. ред.

полнения операции инкремента или декремента. То есть для выражения эта операция как бы не существует, она выполняется только для операнда. Например,

```
x = 10;  
y = ++x;
```

присваивает *y* значение 11. Однако если написать

```
x = 10;  
y = x++;
```

то переменной *y* будет присвоено значение 10. В обоих случаях *x* присвоено значение 11, разница только в том, когда именно это случилось, до или после присваивания значения переменной *y*.

Большинство компиляторов С генерируют для инкремента и декремента очень быстрый, эффективный объектный код, значительно лучший, чем для соответствующих операторов присваивания. Поэтому везде, где это возможно, рекомендуется использовать инкремент и декремент.

Приоритет выполнения арифметических операторов следующий:

Наивысший	++ --
	- (унарный минус)
	* / %
Наинизший	+ -

Операции с одинаковым приоритетом выполняются слева направо. Используя круглые скобки, можно изменить порядок вычислений. В языке С круглые скобки интерпретируются компилятором так же, как и в любом другом языке программирования: они как бы придают операции (или последовательности операций) наивысший приоритет.

Операции сравнения и логические операции

Операции *сравнения* — это операции, в которых значения двух переменных сравниваются друг с другом. *Логические* же операции реализуют средствами языка С операции формальной логики. Между логическими операциями и операциями сравнения существует тесная связь: результаты операций *сравнения* часто являются операндами *логических* операций.

В операциях сравнения и логических операциях в качестве операндов и результатов операций используются значения ИСТИНА (true) и ЛОЖЬ (false). В языке С значение ИСТИНА представляется любым числом, отличным от нуля. Значение ЛОЖЬ представляется нулем. Результатом операции сравнения или логической операции являются ИСТИНА (true, 1) или ЛОЖЬ (false, 0).

На заметку

Как в С89, так и в С99 значение ИСТИНА представлено любым отличным от нуля числом, а ЛОЖЬ — нулем. В стандарте С99 дополнительно определен тип данных `_Bool`, переменные которого могут принимать значение только 0 или 1. Подробнее см. часть II.

В табл. 2.5 приведен полный список операций сравнения и логических операций. Таблица истинности логических операций имеет следующий вид:

<i>p</i>	<i>q</i>	<i>p && q</i>	<i>p q</i>	! <i>p</i>
0	0	0	0	1
0	1	0	1	1
1	1	1	1	0
1	0	0	1	0

Как операции сравнения, так и логические операции имеют низший приоритет по сравнению с арифметическими. То есть, выражение $10 > 1 + 12$ интерпретируется как $10 > (1 + 12)$. Результат, конечно, равен ЛОЖЬ.

В одном выражении можно использовать несколько операций:

$10 > 5 \ \&\& \ !(10 < 9) \ || \ 3 < 4$

В этом случае результатом будет ИСТИНА.

В языке С не определена операция “исключающего ИЛИ” (exclusive OR, или XOR). Однако с помощью логических операторов несложно написать функцию, выполняющую эту операцию. Результатом операции “исключающее ИЛИ” является ИСТИНА, если и только если один из операндов (но не оба) имеют значение ИСТИНА. В следующем примере функция `xor()` возвращает результат операции “исключающее ИЛИ”, а операндами служат аргументы функции:

```
#include <stdio.h>
int xor(int a, int b);

int main(void)
{
    printf("%d", xor(1, 0));
    printf("%d", xor(1, 1));
    printf("%d", xor(0, 1));
    printf("%d", xor(0, 0));

    return 0;
}

/* Выполнение логической операции
   исключающее ИЛИ над двумя аргументами */
int xor(int a, int b)
{
    return (a || b) && !(a && b);
}
```

Таблица 2.5. Операции сравнения и логические операции

<i>Операторы сравнения</i>	
<i>Оператор</i>	<i>Операция</i>
>	Больше чем
>=	Больше или равно
<	Меньше чем
<=	Меньше или равно
==	Равно
!=	Не равно
<i>Логические операторы</i>	
<i>Оператор</i>	<i>Операция</i>
&&	И
	ИЛИ
!	НЕ, отрицание

Ниже приведен приоритет логических операций:

Наивысший	!
	> >= < <=
	== !=
	&&
Наинизший	

Как и в арифметических выражениях, для изменения порядка выполнения операций сравнения и логических операций можно использовать круглые скобки. Например, выражение:

```
!0 && 0 || 0
```

равно ЛОЖЬ. Однако, если добавить скобки как показано ниже, то результатом будет ИСТИНА:

```
!(0 && 0) || 0
```

Необходимо помнить, что результатом любой операции сравнения или логической операции есть 0 или 1. Поэтому следующий фрагмент программы является правильным и в результате его выполнения будет напечатано 1.

```
int x;  
x = 100;  
printf("%d", x>10);
```

Поразрядные операции

В отличие от многих других языков программирования, в С определен полный набор *поразрядных операций*¹. Это обусловлено тем, что С был задуман как язык, призванный во многих приложениях заменить ассемблер, который способен оперировать битами данных. *Поразрядные операции* — это тестирование (проверка), сдвиг или присвоение значений отдельным битам данных. Эти операции осуществляются над ячейками памяти, содержащими данные типа `char` или `int`. Данные типа `float`, `double`, `long double`, `void` или другие более сложные не могут участвовать в поразрядных операциях. В табл. 2.6 приведен полный список знаков поразрядных операций, выполняемых над отдельными разрядами (битами) операндов.

Таблицы истинности логических операций и поразрядных операций И, ИЛИ, НЕ совпадают. Отличие лишь в том, что поразрядные операции выполняются над отдельными разрядами (битами) операндов. Операция “исключающее ИЛИ” имеет следующую таблицу истинности:

p	q	$p \wedge q$
0	0	0
1	0	1
1	1	0
0	1	1

Как показано в таблице, результат операции “исключающее ИЛИ” равен ИСТИНА если и только если один из операндов равен 1, иначе результат будет равен ЛОЖЬ.

Наиболее часто поразрядные операции применяются при программировании драйверов устройств, таких как модемы, а также процедур, выполняющих операции над файлами, и стандартных программ обслуживания принтера. В них поразрядные операции используются для маскирования определенных битов, например, бита контроля

¹ Называются также *битовыми*, *побитовыми* и *логическими операциями*. — Прим. ред.

четности¹. (Этот бит служит для проверки правильности остальных битов в байте. Чаще всего это бит старшего разряда в каждом байте.)

Таблица 2.6. Поразрядные операции

Оператор	Операция
&	И
	ИЛИ
^	исключающее ИЛИ
~	НЕ (отрицание, дополнение к 1)
>>	Сдвиг вправо
<<	Сдвиг влево

Операция И может быть использована для *очистки бита*². Иными словами, для гашения бита используется следующее свойство операции И: если бит одного из операндов равен 0, то соответствующий бит результата будет равен 0 независимо от состояния этого бита во втором операнде. Например, следующая функция читает символ из порта модема и обнуляет бит контроля четности:

```
char get_char_from_modem(void)
{
    char ch;

    ch = read_modem(); /* чтение символа из
                        порта модема */
    return (ch & 127);
}
```

Бит контроля четности, находящийся в 8-м разряде байта, обнуляется с помощью операции И. При этом в качестве второго операнда выбирается число, имеющее 1 в разрядах от 1 до 7, и 0 в 8-м разряде. Именно таким числом и является 127, поскольку все биты двоичного представления числа 127, кроме старшего, равны 1. В силу указанного свойства операции И операция `ch & 127` оставляет все биты, кроме старшего, без изменения, а старший обнуляет:

Бит контроля четности

↓	
11000001	переменная <code>ch</code> содержит символ 'A' с битом четности
01111111	двоичное представление числа 127
& —————	поразрядная операция И
01000001	символ 'A' с обнуленным битом контроля четности

¹ *Бит контроля четности* называется также контрольным двоичным разрядом четности, контрольным разрядом четности, проверочным двоичным разрядом четности, проверочным разрядом четности, битом четности, разрядом четности, контрольным битом и битом контроля на четность. Это дополнительный бит, который добавляется к группе (обычно из семи) битов. Передающее устройство устанавливает значение бита четности равным нулю или единице так, чтобы сумма битов в каждом байте всегда была четной или нечетной в зависимости от выбора типа проверки — на четность или нечетность. Невыполнение условия такой проверки на приемном конце линии означает искажение по крайней мере одного бита при передаче. При обнаружении ошибки принимающее устройство делает запрос на повтор данных. Иными словами, этот бит, добавляемый к данным для контроля их верности таким образом, чтобы сумма двоичных единиц, составляющих данное, включая единицу контрольного бита, всегда была четной (либо всегда нечетной). — *Прим. ред.*

² *Очищение бита* — гашение, т.е. занесение нуля. — *Прим. ред.*

Поразрядная операция ИЛИ, являющаяся двойственной операции И, применяется для установки необходимых битов в 1. В следующем примере выполняется операция $128 \mid 3$:

10000000	двоичное представление числа 128
00000011	двоичное представление числа 3
	поразрядная операция ИЛИ
10000011	результат

Операция исключающего ИЛИ (XOR) устанавливает бит результата в 1, если соответствующие биты операндов различны. В следующем примере выполняется операция $127 \wedge 120$:

01111111	двоичное представление числа 127
01111000	двоичное представление числа 120
^	поразрядная операция XOR
00000111	результат

Необходимо помнить, что результат логической операции всегда равен 0 или 1. В то же время результатом поразрядной операции может быть любое значение, которое, как видно из предыдущих примеров, не обязательно равно 0 или 1.

Поразрядные операторы сдвига \gg и \ll сдвигают все биты переменной вправо или влево. Общая форма оператора сдвига вправо:

переменная \gg *количество_разрядов*

Общая форма оператора сдвига влево:

переменная \ll *количество_разрядов*

Во время сдвига битов в один конец числа, другой конец заполняется нулями. Но если число типа `signed int` отрицательно, то при сдвиге вправо левый конец заполняется единицами, так что знак числа сохраняется. Необходимо отметить различие между сдвигом и циклическим сдвигом. При циклическом сдвиге биты, сдвигаемые за пределы операнда, появляются на другом конце операнда. А при сдвиге вышедшие за границу биты теряются.

Поразрядные операции сдвига очень полезны при декодировании выходов внешних устройств, например таких, как цифро-аналоговые преобразователи, а также при считывании информации о статусе устройств. Побитовые операторы сдвига могут быстро умножать и делить целые числа. Как показано в табл. 2.7, сдвиг на один бит вправо делит число на 2, а на один бит влево — умножает на 2. Следующая программа иллюстрирует применение операторов сдвига:

```
/* Пример применения операторов сдвига */
#include <stdio.h>

int main(void)
{
    unsigned int i;
    int j;

    i = 1;

    /* сдвиг влево */
    for(j=0; j<4; j++) {
        i = i << 1; /* сдвиг i влево на 1 разряд, что
                     равносильно умножению на 2 */
        printf("Сдвиг влево на %d разр.: %d\n", j, i);
    }

    /* сдвиг вправо */
```

```

for(j=0; j<4; j++) {
    i = i >> 1; /* сдвиг i вправо на 1 разряд, что
                равносильно делению на 2 */
    printf("Сдвиг вправо на %d разр.: %d\n", j, i);
}

return 0;
}

```

Поразрядная операция отрицания (дополнения) \sim инвертирует состояние каждого бита операнда. То есть, 0 преобразует в 1, а 1 — в 0.

Поразрядные операции часто используются в процедурах кодирования. Прodelав с дисковым файлом некоторые поразрядные операции, его можно сделать нечитаемым. Простейший способ сделать это — применить операцию отрицания к каждому биту:

Исходный байт 0010100

После 1-го отрицания 1101011

После 2-го отрицания 0010100

Обратите внимание, при последовательном применении 2-х отрицаний результатом всегда будет исходное число. Таким образом, 1-е отрицание кодирует состояние байта, а 2-е — декодирует.

Таблица 2.7. Умножение и деление операторами сдвига

<i>unsigned char x</i>	<i>x после операции</i>	<i>значение x</i>
$x = 7;$	00000111	7
$x = x << 1;$	00001110	14
$x = x << 3;$	01110000	112
$x = x << 2;$	11000000	192
$x = x >> 1;$	01100000	96
$x = x >> 2;$	00011000	24

Каждый сдвиг влево умножает на 2. Потеря информации произошла после операции $x << 2$ в результате сдвига за левую границу.

Каждый сдвиг вправо делит на 2. Сдвиг вправо потерянную информацию не восстановил.

В следующем примере оператор отрицания используется в функции шифрования символа:

```

/* Простейшая процедура шифрования */
char encode(char ch)
{
    return(~ch); /* операция отрицания */
}

```

Конечно, взломать такой шифр не представляет труда.

Операция ?

В языке C определен мощный и удобный оператор, который часто можно использовать вместо оператора вида if-then-else. Речь идет о тернарном операторе $?$, общий вид которого следующий:

Выражение1 ? Выражение2 : Выражение3;

Обратите внимание на использование двоеточия. Оператор ? работает следующим образом: сначала вычисляется *Выражение1*; если оно истинно, то вычисляется *Выражение2* и его значение присваивается всему выражению; если *Выражение1* ложно, то вычисляется *Выражение3* и всему выражению присваивается его значение. В примере

```
x = 10;  
y = x > 9 ? 100 : 200;
```

переменной *y* будет присвоено значение 100. Если бы *x* было меньше 9, то переменной *y* было бы присвоено значение 200. Эту же процедуру можно написать, используя оператор *if-else*:

```
x = 10;  
if (x > 9) y = 100;  
else y = 200;
```

Более подробно оператор ? обсуждается в главе 3 в связи с условными операторами.

Операции получения адреса (&) и раскрытия ссылки (*)

Указатель — это адрес объекта в памяти. *Переменная типа “указатель”* (или просто *переменная-указатель*) — это специально объявленная переменная, в которой хранится указатель на переменную определенного типа. В языке C указатели служат мощнейшим средством создания программ и широко используются для самых разных целей. Например, с их помощью можно быстро обратиться к элементам массива или дать функции возможность модифицировать свои аргументы. Указатели широко используются для связи элементов в списках, в двоичных деревьях и в других динамических структурах данных. Глава 5 полностью посвящена указателям. В данной главе коротко рассматриваются два оператора, использующиеся для работы с указателями.

Первый из них — *оператор &*, это унарный оператор, возвращающий адрес операнда в памяти¹. (Унарной операцией называется операция, имеющая только один операнд.) Например, оператор

```
m = &count;
```

записывает в переменную *m* адрес переменной *count*. Этот адрес представляет собой адрес ячейки памяти компьютера, в которой размещена переменная. Адрес и значение переменной — совершенно разные понятия. Выражение “&переменная” означает “адрес переменной”. Следовательно, инструкция *m = &count;* означает: “Переменной *m* присвоить адрес, по которому расположена переменная *count*;”.

Допустим, переменная *count* расположена в памяти в ячейке с адресом 2000, а ее значение равно 100. Тогда в предыдущем примере переменной *m* будет присвоено значение 2000.

Второй рассматриваемый *оператор ** является двойственным (дополняющим) по отношению к &². Оператор *** является унарным оператором, он возвращает значение объекта, расположенного по указанному адресу. Операндом для *** служит адрес объекта (переменной). Например, если переменная *m* содержит адрес переменной *count*, то оператор

```
q = *m;
```

записывает значение переменной *count* в переменную *q*. В нашем примере переменная *q* получит значение 100, потому что по адресу 2000 записано число 100, причем этот адрес

¹ Оператор & называется также *оператором получения (взятия) адреса*. — Прим. ред.

² Оператор * называется также *оператором косвенности, оператором раскрытия ссылки и оператором разыменования адреса*. — Прим. ред.

записан в переменной `m`. Выражение “* адрес” означает “по адресу”. Наш фрагмент программы можно прочесть как “`q` получает значение, расположенное по адресу `m`”.

К сожалению, символ операции раскрытия ссылки совпадает с символом операции умножения, а символ операции получения адреса — с символом операции поразрядного И. Необходимо помнить, что эти операторы не имеют никакого отношения друг к другу. Операторы `*` и `&` имеют более высокий приоритет, чем любая арифметическая операция, кроме унарного минуса, имеющего такой же приоритет.

Если переменная является указателем, то в объявлении перед ее именем нужно поставить символ `*`, он сообщит компилятору о том, что это указатель на переменную данного типа. Например, объявление указателя на переменную типа `char` записывается так:

```
char *ch;
```

Необходимо понимать, что `ch` — это не переменная типа `char`, а указатель на переменную данного типа, это совершенно разные вещи. Тип данных, на который указывает указатель (в данном случае это `char`), называется *базовым типом* указателя¹. Сам указатель является переменной, содержащей адрес объекта базового типа. Компилятор учтет размер указателя в архитектуре компьютера и выделит для него необходимое количество байтов, чтобы в указатель поместился адрес. Базовый тип указателя определяет тип объекта, хранящегося по этому адресу.

В одном операторе объявления можно одновременно объявить и указатель, и переменную, не являющуюся указателем. Например, оператор

```
int x, *y, count;
```

объявляет `x` и `count` как переменные целого типа, а `y` — как указатель на переменную целого типа.

В следующей программе операторы `*` и `&` используются для записи значения 10 в переменную `target`. Программа выведет значение 10 на экран.

```
#include <stdio.h>

int main(void)
{
    int target, source;
    int *m;

    source = 10;
    m = &source;
    target = *m;

    printf("%d", target);

    return 0;
}
```

Операция определения размера `sizeof`

Унарная операция `sizeof`, выполняемая во время компиляции программы, позволяет определить длину операнда в байтах. Например, если компилятор для чисел типа `int` отводит 4 байта, а для чисел типа `double` — 8, то следующая программа напечатает 8 4.

```
double f;

printf("%d ", sizeof f);
printf("%d ", sizeof(int));
```

¹ Иногда называется также *основным* или *исходным* типом. — Прим. ред.

Необходимо помнить, что для вычисления размера типа переменной имя типа должно быть заключено в круглые скобки. Имя переменной заключать в скобки не обязательно, но ошибки в этом не будет.

В языке С определяется (с помощью спецификатора класса памяти `typedef`) специальный тип `size_t`, приблизительно соответствующий целому числу без знака. Результат операции `sizeof` имеет тип `size_t`. Но практически его можно использовать везде, где допустимо использование целого числа без знака.

Оператор `sizeof` очень полезен для улучшения переносимости программ, так как переносимость существенно зависит от размеров встроенных типов данных. Для примера рассмотрим программу, работающую с базой данных, в которой необходимо хранить шесть целых чисел в одной записи. Если эта программа предназначена для работы на многих компьютерах, ни в коем случае нельзя полагаться на то, что размер целого числа на всех компьютерах будет один и тот же. В программе следует определять размер целого, используя оператор `sizeof`. Соответствующая программа имеет следующий вид:

```
/* Запись шести целых чисел в дисковый файл */
void put_rec(int rec[6], FILE *fp)
{
    int len;

    len = fwrite(rec, sizeof(int)*6, 1, fp);
    if(len != 1) printf("Ошибка при записи");
}
```

Приведенная функция `put_rec()` компилируется и выполняется правильно в любой среде, в том числе на 16- и 32-разрядных компьютерах.

И в заключение: оператор `sizeof` выполняется во время трансляции, его результат в программе рассматривается как константа.

Оператор последовательного вычисления: оператор “запятая”

Оператор “запятая”¹ связывает воедино несколько выражений. При вычислении левой части оператора “запятая” всегда подразумевается, что она имеет тип `void`. Это значит, что выражение, стоящее справа после оператора “запятая”, является значением всего разделенного запятыми выражения. Например, оператор

```
x = (y=3, y+1);
```

сначала присваивает `y` значение 3, а затем присваивает `x` значение 4. Скобки здесь обязательны, потому что приоритет оператора “запятая” меньше, чем оператора присваивания.

В операторе “запятая” выполняется последовательность операций. Если этот оператор стоит в правой части оператора присваивания, то его результатом всегда является выражение, стоящее последним в списке.

Оператор доступа к члену структуры (оператор `.` (точка)) и оператор доступа через указатель `->` (оператор стрелка)

В языке С операторы `.` (точка) и `->` (стрелка) обеспечивают доступ к элементам структур и объединений. *Структуры и объединения* — это составные типы данных, в которых под одним именем хранятся многие объекты. (Структуры и объединения подробно рассматриваются в главе 7.)

¹ Чаще встречается написание без кавычек: *оператор запятая*. Мы пишем кавычки лишь для того, чтобы новичкам было легче воспринимать несколько непривычное для них название оператора. — *Прим. ред.*

Оператор точка используется для прямой ссылки на элемент структуры или объединения, т.е. перед точкой стоит имя структуры, а после — имя элемента структуры. Оператор стрелка используется с указателем на структуру или объединение, т.е. перед стрелкой стоит указатель на структуру. Например, во фрагменте программы

```
struct employee
{
    char name[80];
    int age;
    float wage;
} emp;

struct employee *p = &emp; /* адрес emp заносится в p */
для присвоения члену wage значения 123.33 необходимо записать
emp.wage = 123.33;
```

То же самое можно сделать, используя указатель на структуру:

```
p->wage = 123.33;
```

Операторы [] и ()

Круглые скобки являются оператором, повышающим приоритет выполнения операций, которые в них заключены. Квадратные скобки служат для индексации массива (массивы подробно рассматриваются в главе 4). Если в программе определен массив, то выражение в квадратных скобках представляет собой индекс массива. Например, в программе

```
#include <stdio.h>
char s[80];

int main(void)
{
    s[3] = 'X';
    printf("%c", s[3]);

    return 0;
}
```

значение 'X' сначала присваивается четвертому элементу массива (в C элементы массива нумеруются с нуля), затем этот элемент выводится на экран.

Сводка приоритетов операций

В табл. 2.8 приведены приоритеты всех операций, определенных в C. Необходимо помнить, что все операторы, кроме унарных и “?”, связывают (присоединяют, ассоциируют) свои операнды слева направо. Унарные операторы (*, &, -) и “?” связывают (присоединяют, ассоциируют) свои операнды справа налево.

Таблица 2.8. Приоритеты операций в языке C

Наивысший	() [] -> .
	! ~ ++ -- (type) * & sizeof
	* / %
	+ -
	<< >>
	< <= > >=

	== !=
	&
	^
	&&
	?:
Наинизший	= += -= *= /= и т.д.



Выражения

Выражения состоят из операторов, констант, функций и переменных. В языке C выражением является любая правильная последовательность этих элементов. Большинство выражений в языке C по форме очень похожи на алгебраические, часто их и пишут, руководствуясь правилами алгебры. Однако здесь необходимо быть внимательным и учитывать специфику выражений в языке C.

Порядок вычислений

Порядок вычисления подвыражений в выражениях языка C не определен. Компилятор может самостоятельно перестроить выражение с целью создания оптимального объектного кода. Это значит, что программист не может полагаться на определенную последовательность вычисления подвыражений. Например, при вычислении выражения

```
x = f1() + f2();
```

нет никаких гарантий того, что функция `f1()` будет вызвана перед вызовом `f2()`.

Преобразование типов в выражениях

Если в выражении встречаются переменные и константы разных типов, они преобразуются к одному типу. Компилятор преобразует “меньший” тип в “большой”. Этот процесс называется *продвижением типов (type promotion)*. Сначала все переменные типов `char` и `short int` автоматически продвигаются в `int`. Это называется *целочисленным расширением*. (В C99 целочисленное расширение может также завершиться преобразованием в `unsigned int`.) После этого все остальные операции выполняются одна за другой, как описано в приведенном ниже алгоритме преобразования типов:

```
IF операнд имеет тип long double
THEN второй операнд преобразуется в long double
ELSE IF операнд имеет тип double
THEN второй операнд преобразуется в double
ELSE IF операнд имеет тип float
THEN второй операнд преобразуется в float
ELSE IF операнд имеет тип unsigned long
THEN второй операнд преобразуется в unsigned long
ELSE IF операнд имеет тип long
THEN второй операнд преобразуется в long
ELSE IF операнд имеет тип unsigned int
THEN второй операнд преобразуется в unsigned int
```

Для тех, кто еще не знаком с общей формой оператора IF, приводим более русифицированную запись алгоритма¹:

```
ЕСЛИ операнд имеет тип long double
ТО второй операнд преобразуется в long double
ИНАЧЕ ЕСЛИ операнд имеет тип double
ТО второй операнд преобразуется в double
ИНАЧЕ ЕСЛИ операнд имеет тип float
ТО второй операнд преобразуется в float
ИНАЧЕ ЕСЛИ операнд имеет тип unsigned long
ТО второй операнд преобразуется в unsigned long
ИНАЧЕ ЕСЛИ операнд имеет тип long
ТО второй операнд преобразуется в long
ИНАЧЕ ЕСЛИ операнд имеет тип unsigned int
ТО второй операнд преобразуется в unsigned int
```

Кроме того, действует следующее правило: если один из операндов имеет тип long, а второй — unsigned int, притом значение unsigned int не может быть представлено типом long, то оба операнда преобразуются в unsigned long.

На заметку

Описание правил целочисленного расширения в C99 см. в части II.

После выполнения приведенных выше преобразований оба операнда относятся к одному и тому же типу, к этому типу относится и результат операции.

Рассмотрим пример преобразования типов, приведенный на рис. 2.2. Сначала символ ch преобразуется в целое число. Результат операции ch/i преобразуется в double, потому что результат f*d имеет тип double. Результат операции f+i имеет тип float, потому что f имеет тип float. Окончательный результат имеет тип double.

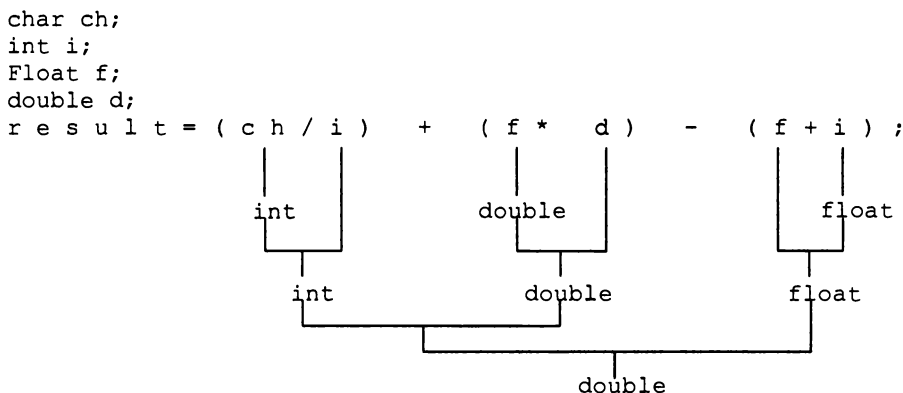


Рис. 2.2. Пример преобразования типов

Явное преобразование типов: операция приведения типов

Программист может “принудительно” преобразовать значение выражения к нужному ему типу, используя операцию *приведения типов*. Общая форма оператора явного приведения типа:

¹ Добавлено редактором русского перевода. — Прим. ред.

(тип) выражение

Здесь *тип* — это любой поддерживаемый тип данных. Например, следующая запись преобразует значение выражения $x/2$ к типу `float`:

```
(float) x/2
```

Явное преобразование типа — это операция. Оператор приведения типа является унарным и имеет тот же приоритет, что и остальные унарные операторы.

Иногда приведение типов может быть весьма полезным. Допустим, целую переменную нужно использовать как параметр цикла, притом в вычислении участвует и дробная часть числа. В следующем примере показано, как с помощью приведения можно сохранить точность:

```
#include <stdio.h>

int main(void) /* печать i и i/2 с дробной частью */
{
    int i;

    for(i=1; i<=100; ++i)
        printf("%d / 2 равно: %f\n", i, (float) i / 2);
    return 0;
}
```

Без операции приведения `(float)` выполнялось бы целочисленное деление. Дробная часть результата выводится благодаря приведению типа переменной `i`.

Пробелы и круглые скобки

Для повышения удобочитаемости программы при записи выражений можно использовать пробелы и символы табуляции. Например, следующие два оператора эквивалентны:

```
x=10/y~(127/x)/
x = 10 / y ~(127/x) ;
```

Лишние скобки, если они не изменяют приоритет операций, не приводят к ошибкам и не замедляют вычисление выражения. Дополнительные скобки часто используют для прояснения порядка вычислений. В следующем примере 2-я строка читается значительно легче:

```
x = y/3-34*temp+127;
x= (y/3) - (34*temp) + 127;
```

Полный
справочник по



Глава 3

Операторы

Оператор — это часть программы, которая может быть выполнена отдельно¹. Это означает, что оператор определяет некоторое действие. В языке С существуют следующие группы операторов:

- Условные операторы
- Операторы цикла
- Операторы безусловного перехода
- Метки
- Операторы-выражения
- Блоки

К условным относятся операторы `if` и `switch`. Иногда их также называют *операторами условного перехода*. Операторы цикла — это `while`, `for` и `do-while`. К операторам безусловного перехода относятся `break`, `continue`, `goto` и `return`. К меткам относятся операторы `case`, `default` (рассматриваются в разделе “Оператор выбора — `switch`”) и собственно метки (рассматриваются в разделе “Оператор `goto`”). Операторы-выражения — это операторы, состоящие из допустимых выражений. Блок представляет собой фрагмент текста программы, обрамленный фигурными скобками `{}`. Блок иногда называют *составным оператором*.

Так как во многих операторах применяются условные выражения, их рассмотрение начнем с краткой характеристики значений **ИСТИНА** и **ЛОЖЬ**.

Логические значения **ИСТИНА** (True) и **ЛОЖЬ** (False) в языке С

При выполнении многих операторов языка С вычисляются значения условных выражений и в зависимости от полученного значения выбирается та или иная ветвь вычислительного процесса. Условное выражение может принимать одно из двух значений: **ИСТИНА** или **ЛОЖЬ**. В языке С значение **ИСТИНА** представлено любым ненулевым значением, включая отрицательные числа. Значение **ЛОЖЬ** всегда представлено нулем. Такое представление логических значений **ИСТИНА** и **ЛОЖЬ** позволяет весьма эффективно программировать многие процедуры.

Условные операторы

В языке С существуют два условных оператора: `if` и `switch`. При определенных обстоятельствах оператор `?` является альтернативой оператора `if`.

Оператор `if`

Общая форма оператора `if` следующая:

```
if (выражение) оператор;  
else оператор;
```

Здесь *оператор* может быть только одним оператором, блоком операторов или отсутствовать (пустой оператор). Фраза `else` может вообще отсутствовать.

¹ Операторы в указанном смысле в языке С называются также инструкциями, а иногда и командами. В других языках операторы могут называться также предложениями (КОБОЛ). — *Прим. ред.*

Если *выражение* истинно¹ (т.е. принимает любое значение, отличное от нуля), то выполняется оператор или блок операторов, следующий за `if`. В противном случае выполняется оператор (или блок операторов), следующий за `else` (если эта фраза присутствует). Необходимо помнить, что выполняется или оператор, связанный с `if`, или с `else`, но оба — никогда!

Условное выражение, входящее в `if`, должно иметь скалярный результат. Это значит, что результатом должно быть целое число, символ, указатель или число с плавающей точкой, но им не может быть массив или структура. (В Стандарте C99 тип `_Bool` также является скалярным, поэтому значение этого типа может использоваться в условии оператора `if`.) В выражении-условии оператора `if` результат плавающего типа используется редко, потому что это существенно замедляет вычислительный процесс. Объясняется это тем, что для выполнения операций над *плавающими операндами*² необходимо выполнить больше команд процессора, чем для выполнения операций над целыми числами или символами.

В следующей программе иллюстрируется использование оператора `if`. В ней запрограммирована очень простая игра “угадай магическое число”. Если играющий угадал число, на экран выводится сообщение `**Верно**`. Программа генерирует “магическое число” с помощью стандартного генератора случайных чисел `rand()`. Генератор возвращает случайное число в диапазоне между 0 и `RAND_MAX` (обычно это число не меньше 32767). Функция `rand()` объявлена в заголовочном файле `stdlib.h`.

```
/* Магическое число, программа N1 */
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    int magic; /* магическое число */
    int guess; /* попытка игрока */

    magic = rand(); /* генерация магического числа */

    printf("Угадай магическое число: ");
    scanf("%d", &guess);

    if(guess == magic) printf("**Верно**");

    return 0;
}
```

В следующей версии программы для игры в “магическое число” иллюстрируется использование оператора `else`. В этой версии выводится дополнительное сообщение в случае ложного ответа.

```
/* Магическое число, программа N2 */
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    int magic; /* магическое число */
```

¹ Иногда в этом случае говорят, что *выражение* принимает значение ИСТИНА, которое в языке C может быть представлено, как мы помним, любым целым числом, отличным от нуля. — Прим. ред.

² Плавающие операнды называются также операндами в формате с плавающей точкой, вещественными операндами. — Прим. ред.

```

int guess; /* попытка игрока */

magic = rand(); /* генерация магического числа */

printf("Угадай магическое число: ");
scanf("%d", &guess);

if(guess == magic) printf("***Верно** ");
else printf("***Неверно** ");

return 0;
}

```

Вложенные условные операторы if

Оператор if является вложенным, если он вложен, т.е. находится внутри другого оператора if или else. В практике программирования вложенные условные операторы используются довольно часто. Во вложенном условном операторе фраза else всегда ассоциирована с ближайшим if в том же блоке, если этот if не ассоциирован с другой фразой else. Например:

```

if(i)
{
    if(j) dosomething1();
    if(k) dosomething2(); /* этот if */
    else dosomething3(); /* ассоциирован с этим else */
}
else dosomething4(); /* ассоциирован с if(i) */

```

Последняя фраза else не ассоциирована с if(j) потому, что она находится в другом блоке. Эта фраза else ассоциирована с if(i). Внутренняя фраза else ассоциирована с if(k), потому что этот if — ближайший.

Стандарт C89 допускает 15 уровней вложенности условных операторов, C99 — 127 уровней. В настоящее время большинство компиляторов допускают значительно большее количество уровней вложенности. Однако на практике необходимость в глубине вложенности, большей, чем несколько уровней, возникает довольно редко, так как увеличение глубины вложенности быстро запутывает программу и делает ее нечитаемой.

В следующем примере вложенный оператор if используется в модернизированной программе для игры в магическое число. С его помощью играющий получает сообщение о характере ошибки:

```

/* Магическое число, программа N3 */
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    int magic; /* магическое число */
    int guess; /* попытка игрока */

    magic = rand(); /* генерация магического числа */

    printf("Угадай магическое число: ");
    scanf("%d", &guess);

    if(guess == magic){

```

```

    printf("***Верно***");
    printf("магическое число равно %d\n", magic);
}

else {
    printf("***Неверно, ");
    if(guess > magic) printf("слишком большое.\n");
    /* вложенный if */
    else printf("слишком малое.\n");
}

return 0;
}

```

Лестница if-else-if

В программах часто используется конструкция, которую называют *лестницей if-else-if*¹. Общая форма лестницы имеет вид

```

if (выражение) оператор;
else
    if (выражение) оператор;
    else
        if (выражение) оператор;
        .
        .
        .
    else оператор;

```

Работает эта конструкция следующим образом. Условные выражения операторов if вычисляются сверху вниз. После выполнения некоторого условия, т.е. когда встретится выражение, принимающее значение ИСТИНА, выполняется ассоциированный с этим выражением оператор, а оставшаяся часть лестницы пропускается. Если все условия ложны, то выполняется оператор в последней фразе else, а если последняя фраза else отсутствует, то в этом случае не выполняется ни один оператор.

Недостаток предыдущей записи лестницы состоит в том, что с ростом глубины вложенности увеличивается количество отступов в строке. Это становится неудобным с технической точки зрения. Поэтому лестницу if-else-if обычно записывают так:

```

if(выражение)
    оператор;
else if(выражение)
    оператор;
else if(выражение)
    оператор;
.
.
.
else
    оператор;

```

Используя лестницу if-else-if, программу для игры в “магическое число” можно записать так:

¹ Называется также *структурой выбора* или *конструкцией условного перехода*. — Прим. ред.


```

/* Магическое число, программа N4 */
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    int magic; /* магическое число */
    int guess; /* попытка игрока */

    magic = rand(); /* генерация магического числа */

    printf("Угадай магическое число: ");
    scanf("%d", &guess);

    if(guess == magic){
        printf("***Верно** ");
        printf("Магическое число равно %d\n", magic);
    }

    else if(guess > magic)
        printf("Неверно, слишком большое");
    else
        printf("Неверно, слишком малое");

    return 0;
}

```

Оператор “?”, альтернативный условному

Оператор ? можно использовать вместо оператора if-else, записанного в форме

if(условие) переменная = выражение;
else переменная = выражение;

Оператор ? является *тернарным*, потому что он имеет три операнда. Его общая форма следующая:

Выражение1 ? Выражение2 : Выражение3;

Обратите внимание на использование и расположение двоеточия.

Результат операции ? определяется следующим образом. Сначала вычисляется *Выражение1*. Если оно имеет значение ИСТИНА, вычисляется *Выражение2* и его значение становится результатом операции ?. Если *Выражение1* имеет значение ЛОЖЬ, вычисляется *Выражение3* и его значение становится результатом операции ?. Например:

```

x = 10;
y = x>9 ? 100 : 200;

```

В этом примере переменной *y* присваивается значение 100. Если бы *x* было меньше 9, то переменная *y* получила бы значение 200. То же самое можно записать, используя оператор if-else:

```

x = 10;
if (x<9) y = 100;
else y = 200;

```

В следующем примере оператор ? используется для присвоения квадрату числа знака числа. (Само число вводится пользователем.) В этой программе при возведении в квадрат фактически сохраняется знак числа. Например, если пользователь введет 10, это число будет возведено в квадрат и в результате программа напечатает 100, а если

пользователь введет число -10, то оно будет возведено в квадрат и результату будет приписан знак числа; в этом случае будет напечатано -100.

```
#include <stdio.h>

int main(void)
{
    int isqrd, i;

    printf("Введите число: ");
    scanf("%d", &i);

    isqrd = i > 0 ? i*i : -(i*i);

    printf(" %d в квадрате равно %d ", i, isqrd);

    return 0;
}
```

(Обратите внимание, что в результате выполнения данной программы могут быть напечатаны не только верные утверждения. Не всегда компьютеры печатают только правильные результаты, если даже они работают без сбоев! — *Прим. ред.*)

Оператор `?` можно использовать вместо `if-else` не только в операторе присваивания. Как известно, все функции (за исключением имеющих тип результата `void`) возвращают значение. Следовательно, в операторе `?` можно использовать вызовы функций. Когда в выражении встречается вызов функции, она выполняется, а возвращаемое ею значение используется при вычислении выражения. Это значит, что можно выполнить одну или несколько функций путем размещения их вызовов в выражениях оператора `?` в качестве операндов. Например:

```
#include <stdio.h>

int f1(int n);
int f2(void);

int main(void)
{
    int t;

    printf("Введите число: ");
    scanf("%d", &t);

    /* печать соответствующего сообщения */
    t ? f1(t) + f2() : printf("Введен нуль.");
    printf("\n");

    return 0;
}

int f1(int n)
{
    printf("%d ", n);
    return 0;
}

int f2(void)
{
    printf(" введено ");
    return 0;
}
```

Эта программа сначала запрашивает число. При вводе нуля вызывается функция `printf()`, выводящая на экран сообщение введен нуль. При вводе отличного от нуля числа выполняются как `f1()`, так и `f2()`. Обратите внимание на то, что значение выражения `?` в этом примере не присваивается никакой переменной, оно просто отбрасывается.

Следует помнить, что компилятор, пытаясь оптимизировать объектный код, может установить любой порядок вычисления значений операндов. В данном примере это значит, что функции `f1()` и `f2()` выполняются в произвольном порядке и сообщение введено может появиться как до, так и после числа.

Используя оператор `?`, программу для игры в “магическое число” можно переписать следующим образом:

```
/* Магическое число, программа N5 */
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    int magic; /* магическое число */
    int guess; /* попытка игрока */

    magic = rand(); /* генерация магического числа */

    printf("Угадай магическое число: ");
    scanf("%d", &guess);

    if(guess == magic){
        printf("***Верно ** ");
        printf("Магическое число равно %d\n", magic);
    }

    else
        guess > magic ? printf("Слишком большое ") :
                        printf("Слишком малое ");

    return 0;
}
```

В этой программе оператор `?` печатает соответствующее сообщение на основе проверки условия `guess > magic`.

Условное выражение

У начинающих программистов иногда возникают трудности в связи с тем, что в условном (*управляющем*) выражении операторов `if` или `?` могут стоять любые операторы, причем это не обязательно операторы отношения или логические (как в языках Basic или Pascal). В языке C значением результата управляющего выражения являются ИСТИНА или ЛОЖЬ, однако тип результата может быть любым скалярным типом. Считается, что любой ненулевой результат представляет значение ИСТИНА, а нулевой — ЛОЖЬ.

В следующем примере программа считывает с клавиатуры два числа, вычисляет их отношение и выводит его на экран. Оператор `if` используется для того, чтобы избежать деления на нуль, если второе число равно нулю.

```
/* Деление первого числа на второе. */

#include <stdio.h>
```

```

int main(void)
{
    int a, b;

    printf("Введите два числа: ");
    scanf("%d%d", &a, &b);

    if(b) printf("%d\n", a/b);
    else printf("Делить на нуль нельзя.\n");

    return 0;
}

```

Если управляющее выражение *b* равно 0, то его результат представляет значение ЛОЖЬ и выполняется оператор *else*. В противном случае (*b* не равно нулю) результат представляет значение ИСТИНА и выполняется деление чисел.

В последнем примере оператор *if* можно записать так:

```

if (b != 0) printf("%d\n", a/b);

```

Но следует отметить, что такая форма записи избыточна, она может привести к генерации неоптимального кода, кроме того, это считается признаком плохого стиля. Переменная *b* сама по себе представляет значение ИСТИНА или ЛОЖЬ, поэтому сравнивать ее с нулем нет необходимости.

Оператор выбора — switch

Оператор выбора *switch* (часто его называют переключателем) предназначен для выбора ветви вычислительного процесса исходя из значения управляющего выражения. (При этом значение управляющего выражения сравнивается со значениями в списке целых или символьных констант. Если будет найдено совпадение, то выполнится ассоциированный с совпавшей константой оператор.) Общая форма оператора *switch* следующая:

```

switch (выражение) {
    case постоянная1:
        последовательность операторов
        break;
    case постоянная2:
        последовательность операторов
        break;
    case постоянная3:
        последовательность операторов
        break;
    .
    .
    .
    default:
        последовательность операторов;
}

```

Значение *выражения* оператора *switch* должно быть таким, чтобы его можно было выразить целым числом. Это означает, что в управляющем выражении можно использовать переменные целого или символьного типа, но только не с плавающей точкой. Значение управляющего *выражения* по очереди сравнивается с *постоянными* в операторах *case*. Если значение управляющего *выражения* совпадет с какой-то из *постоянных*, управление передается на соответствующую метку *case* и выполняется *последовательность операторов* до оператора *break*. Если оператор *break* отсутствует, выполнение последовательности опе-

раторов продолжается до тех пор, пока не встретится `break` (в другой метке — *Прим. ред.*) или не кончится тело оператора `switch` (т.е. блок, следующий за `switch`). Оператор `default` выполняется в том случае, когда значение управляющего выражения не совпало ни с одной постоянной. Оператор `default` также может отсутствовать. В этом случае при отсутствии совпадений не выполняется ни один оператор.

Согласно Стандарту C89, оператор `switch` может иметь как минимум 257 операторов `case`. Стандарт C99 требует поддержки как минимум 1023 операторов `case`. Если вы пишете программы вручную, такое большое количество операторов вам никогда не понадобится¹. Оператор `case` — это метка, однако он не может быть использован сам по себе, вне оператора `switch`.

Оператор `break` — это один из операторов безусловного перехода. Он может применяться не только в операторе `switch`, но и в циклах, (см. раздел “Операторы цикла”). Когда в теле оператора `switch` встречается оператор `break`, программа выходит из оператора `switch` и выполняет оператор, следующий за фигурной скобкой `}` оператора `switch`.

Об операторе `switch` очень важно помнить следующее:

- Оператор `switch` отличается от `if` тем, что в нем управляющее выражение проверяется только на *равенство* с постоянными, в то время как в `if` проверяется любой вид отношения или логического выражения.
- В одном и том же операторе `switch` никакие два оператора `case` не могут иметь равных постоянных. Конечно, если один `switch` вложен в другой, в их операторах `case` могут быть совпадающие постоянные.
- Если в управляющем выражении оператора `switch` встречаются символьные константы, они автоматически преобразуются к целому типу по принятым в языке C правилам приведения типов.

Оператор `switch` часто используется для обработки команд с клавиатуры, например, при выборе пунктов меню. В следующем примере программа выводит на экран меню проверки правописания и вызывает соответствующую процедуру:

```
void menu(void)
{
    char ch;

    printf("1. Проверка правописания\n");
    printf("2. Коррекция ошибок\n");
    printf("3. Вывод ошибок\n");
    printf("Для пропуска нажмите любую клавишу\n");
    printf("    Введите ваш выбор: ");

    ch = getchar(); /* чтение клавиши */

    switch(ch) {
        case '1':
            check_spelling();
            break;
        case '2':
            correct_errors();
            break;
        case '3':
            display_errors();
            break;
        default:
```

¹ Если же для генерации программ вы используете макрогенераторы или генераторы компиляторов, например, уасс или `lex`, то на данное ограничение следует обратить внимание. — *Прим. ред.*

```

    printf("Не выбрана ни одна опция.");
}
}

```

С точки зрения синтаксиса, присутствие операторов `break` внутри `switch` не обязательно. Они прерывают выполнение последовательности операторов, ассоциированных с данной константой. Если оператор `break` отсутствует, то выполняется следующий оператор `case`, пока не встретится очередной `break`, или не будет достигнут конец тела оператора `switch`. Например, в функции `inp_handler()` (обработчик ввода драйвера) для упрощения программы несколько операторов `break` опущено, поэтому выполняются сразу несколько операторов `case`:

```

/* Обработка значения i */
void inp_handler(int i)
{
    int flag;

    flag = -1;

    switch(i) {
        case 1: /* Эти case имеют общую */
        case 2: /* последовательность операторов */
        case 3:
            flag = 0;
            break;
        case 4:
            flag = 1;
        case 5:
            error(flag);
            break;
        default:
            process(i);
    }
}

```

Приведенный пример иллюстрирует следующие две особенности оператора `switch()`.

Во-первых, оператор `case` может не иметь ассоциированной с ним последовательности операторов. Тогда управление переходит к следующему `case`. В этом примере три первых `case` вызывают выполнение одной и той же последовательности операторов, а именно:

```

flag = 0;
break;

```

Во-вторых, если оператор `break` отсутствует, то выполняется последовательность операторов следующего `case`. Если `i` равно 4, то переменной `flag` присваивается значение 1 и, поскольку `break` отсутствует, выполнение продолжается и вызывается `error(flag)`. Если `i` равно 5, то `error()` будет вызвана со значением переменной `flag`, равным -1, а не 1.

То, что при отсутствии `break` операторы `case` выполняются вместе, позволяет избежать ненужного дублирования операторов¹.

¹ Но представляет собой опасность для забывчивых программистов. — *Прим. ред.*

Вложенные операторы switch

Оператор switch может находиться в теле внешнего по отношению к нему оператора switch. Операторы case внутреннего и внешнего switch могут иметь одинаковые константы, в этом случае они не конфликтуют между собой. Например, следующий фрагмент программы вполне работоспособен:

```
switch(x) {
    case 1:
        switch(y) {
            case 0: printf("Деление на нуль.\n");
                    break;
            case 1: process(x, y);
                    break;
        }
        break;
    case 2:
        .
        .
}
```



Операторы цикла

В языке С, как и в других языках программирования, операторы цикла служат для многократного выполнения последовательности операторов до тех пор, пока выполняется некоторое условие. Условие может быть установленным заранее (как в операторе for) или меняться при выполнении тела цикла (как в while или do-while).

Цикл for

Во всех процедурных языках программирования циклы for очень похожи. Однако в С этот цикл особенно гибкий и мощный.

Общая форма оператора for следующая:

for (*инициализация; условие; приращение*) *оператор*;

Цикл for может иметь большое количество вариаций. В наиболее общем виде принцип его работы следующий. *Инициализация* — это присваивание начального значения переменной, которая называется параметром цикла. *Условие* представляет собой условное выражение, определяющее, следует ли выполнять *оператор* цикла (часто его называют *телом цикла*) в очередной раз. Оператор *приращение* осуществляет изменение параметра цикла при каждой итерации. Эти три оператора (они называются также *секциями* оператора for) обязательно разделяются точкой с запятой. Цикл for выполняется, если выражение *условие* принимает значение ИСТИНА. Если оно хотя бы один раз примет значение ЛОЖЬ, то программа выходит из цикла и выполняется оператор, следующий за телом цикла for.

В следующем примере в цикле for выводятся на экран числа от 1 до 100:

```
#include <stdio.h>

int main(void)
{
    int x;
    for(x=1; x <= 100; x++) printf("%d ", x);
    return 0;
}
```

В этом примере параметр цикла `x` инициализирован числом 1, а затем при каждой итерации сравнивается с числом 100. Пока переменная `x` меньше 100, вызывается функция `printf()` и цикл повторяется. При этом `x` увеличивается на 1 и опять проверяется условие цикла `x <= 100`. Процесс повторяется, пока переменная `x` не станет больше 100. После этого процесс выходит из цикла, а управление передается оператору, следующему за ним. В этом примере параметром цикла является переменная `x`, при каждой итерации она изменяется и проверяется в секции условия цикла.

В следующем примере в цикле `for` выполняется блок операторов:

```
for(x=100; x != 65; x -= 5) {
    z = x*x;
    printf("Квадрат %d равен %d\n", x, z);
}
```

Операции возведения переменной `x` в квадрат и вызова функции `printf()` повторяются, пока `x` не примет значение 65. Обратите внимание на то, что здесь параметр цикла уменьшается, он инициализирован числом 100 и уменьшается на 5 при каждой итерации.

В операторе `for` условие цикла всегда проверяется перед началом итерации. Это значит, что операторы цикла могут не выполняться ни разу, если перед первой итерацией условие примет значение ЛОЖЬ. Например, в следующем фрагменте программы

```
x = 10;
for(y=10; y != x; ++y) printf("%d", y);
printf("%d", y); /* Это единственный printf(),
                который будет выполнен */
```

цикл не выполнится ни разу, потому что при входе в цикл значения переменных `x` и `y` равны. Поэтому условие цикла принимает значение ЛОЖЬ, а тело цикла и оператор *приращение* не выполняются. Переменная `y` остается равной 10, единственный результат работы этой программы — вывод на экран числа 10 в результате вызова функции `printf()`, расположенной вне цикла.

Варианты цикла `for`

В предыдущем разделе рассмотрена наиболее общая форма цикла `for`. Однако в языке C допускаются некоторые его варианты, позволяющие во многих случаях увеличить мощность и гибкость программы.

Один из распространенных способов усиления мощности цикла `for` — применение оператора “запятая” для создания двух параметров цикла. Оператор “запятая” связывает несколько выражений, заставляя их выполняться вместе (см. главу 2). В следующем примере обе переменные (`x` и `y`) являются параметрами цикла `for` и обе инициализируются в этом цикле:

```
for(x=0, y=0; x+y < 10; ++x) {
    y = getchar();
    y = y - '0'; /* Вычитание из y ASCII-кода нуля */
    .
    .
    .
}
```

Здесь запятая разделяет два оператора инициализации. При каждой итерации значение переменной `x` увеличивается, а значение `y` вводится с клавиатуры. Для выполнения итерации как `x`, так и `y` должны иметь определенное значение. Несмотря на то что значение `y` вводится с клавиатуры, оно должно быть инициализировано таким образом, чтобы выполнилось условие цикла при первой итерации. Если `y` не инициали-

зировать, то оно может случайно оказаться таким, что условие цикла примет значение ЛОЖЬ, тело цикла не будет выполнено ни разу.

Следующий пример демонстрирует использование двух параметров цикла. Функция `converge()` копирует содержимое одной строки в другую, начиная с обоих концов строки и кончая в ее середине.

```
/* Демонстрация использования 2-х параметров цикла */
#include <stdio.h>
#include <string.h>

void convergence(char *targ, char *src);

int main(void)
{
    char target[80] = "XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX";

    convergence(target,
        "Это проверка функции converge().");
    printf("Строка-результат: %s\n", target);
    return 0;
}

/* Эта функция копирует содержимое одной строки в
другую, начиная с обоих концов и сходясь посередине */
void convergence(char *targ, char *src)
{
    int i, j;

    printf("%s\n", targ);
    for(i=0, j=strlen(src); i<=j; i++, j--) {
        targ[i] = src[i];
        targ[j] = src[j];
        printf("%s\n", targ);
    }
}
```

Программа выводит на экран следующее:

```
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
ЭXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
ЭтXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX.
ЭтоXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX.
Это XXXXXXXXXXXXXXXXXXXXXXXXXXXXX().
Это пXXXXXXXXXXXXXXXXXXXXXXXXXXXXе().
Это прXXXXXXXXXXXXXXXXXXXXXXXXXXge().
Это проXXXXXXXXXXXXXXXXXXXXrge().
Это провXXXXXXXXXXXXXXXXXXXXerge().
Это провеXXXXXXXXXXXXXXXXXerge().
Это проверXXXXXXXXXXXXnverge().
Это проверкXXXXXXXXXXXXonverge().
Это проверкаXXXXXXXXXXconverge().
Это проверка XXXXXXX converge().
Это проверка фXXXXXи converge().
Это проверка фуXXXXии converge().
Это проверка фунXции converge().
Это проверка функции converge().
Строка-результат: Это проверка функции converge().
```

В функции `convergence()` цикл `for` использует два параметра цикла (`i` и `j`) для индексации строки с противоположных концов. Параметр `i` в цикле увеличивается, а `j` — уменьшается. Итерации прекращаются, когда `i` становится больше `j`. Это обеспечивает копирование всех символов.

Проверка параметра цикла на соответствие некоторому условию не обязательна. Условие может быть любым логическим оператором или оператором отношения. Это значит, что условие выполнения цикла может состоять из нескольких условий, или операторов отношения. Следующий пример демонстрирует применение составного условия цикла для проверки пароля, вводимого пользователем. Пользователю предоставляются три попытки ввода пароля. Программа выходит из цикла, когда использованы все три попытки или когда введен верный пароль.

```
void sign_on(void)
{
    char str[20];
    int x;

    for(x=0; x<3 && strcmp(str, "password")); {
        printf("Пожалуйста, введите пароль: ");
        gets(str);
    }

    if(x == 3) return;
    /* Иначе пользователь допускается. */
}
```

Функция `sign_on()` использует стандартную библиотечную функцию `strcmp()`, которая сравнивает две строки и возвращает 0, если они совпадают.

Следует помнить, что каждая из трех секций оператора `for` может быть любым синтаксически правильным выражением. Эти выражения не всегда каким-либо образом отображают назначение секции. Рассмотрим следующий пример:

```
#include <stdio.h>

int sqrnum(int num);
int readnum(void);
int prompt(void);

int main(void)
{
    int t;

    for(prompt(); t=readnum(); prompt())
        sqrnum(t);

    return 0;
}

int prompt(void)
{
    printf("Введите число ");
    return 0;
}

int readnum(void)
{
    int t;
```

```

    scanf("%d", &t);
    return t;
}

int sqrnum(int num)
{
    printf("%d\n", num*num);
    return num*num;
}

```

Здесь в `main()` каждая секция цикла `for` состоит из вызовов функций, которые предлагают пользователю ввести число и считывают его. Если пользователь ввел 0, то цикл прекращается, потому что тогда условие цикла принимает значение ЛОЖЬ. В противном случае число возводится в квадрат. Таким образом, в этом примере цикла `for` секции инициализации и приращения используются весьма необычно, но совершенно правильно.

Другая интересная особенность цикла `for` состоит в том, что его секции могут быть вообще пустыми, присутствие в них какого-либо выражения не обязательно. В следующем примере цикл выполняется, пока пользователь не введет число 123:

```

for(x=0; x!= 123; ) scanf("%d", &x);

```

Секция приращения оператора `for` здесь оставлена пустой. Это значит, что перед каждой итерацией значение переменной `x` проверяется на неравенство числу 123, а приращения не происходит, оно здесь ненужно. Если с клавиатуры ввести число 123, то условие принимает значение ЛОЖЬ и программа выходит из цикла.

Инициализацию параметра цикла `for` можно сделать за пределами этого цикла, но, конечно, до него. Это особенно уместно, если начальное значение параметра цикла вычисляется достаточно сложно, например:

```

gets(s); /* читает строку в s */
if(*s) x = strlen(s); /* вычисление длины строки */
else x = 10;

for( ; x < 10; ) {
    printf("%d", x);
    ++x;
}

```

В этом примере секция инициализации оставлена пустой, а переменная `x` инициализируется до входа в цикл.

Бесконечный цикл

Для создания бесконечного цикла можно использовать любой оператор цикла, но чаще всего для этого выбирают оператор `for`. Так как в операторе `for` может отсутствовать любая секция, бесконечный цикл проще всего сделать, оставив пустыми все секции. Это хорошо показано в следующем примере:

```

for( ; ; ) printf(" Этот цикл крутится бесконечно ");

```

Если условие цикла `for` отсутствует, то предполагается, что его значение — ИСТИНА. В операторе `for` можно добавить выражения инициализации и приращения, хотя обычно для создания бесконечного цикла используют конструкцию `for(; ;)`.

Фактически конструкция `for(; ;)` не гарантирует бесконечность итераций, потому что в нем может встретиться оператор `break`, вызывающий немедленный выход из цикла. (Подробно оператор `break` рассмотрен в этой главе далее.) В этом случае

выполнение программы продолжается с оператора, следующего за закрывающейся фигурной скобкой цикла `for`:

```
ch = '\0';

for( ; ; ) {
    ch = getchar(); /* считывание символа */
    if (ch == 'A') break; /* выход из цикла */
}

printf(" Вы напечатали 'A'");
```

В данном примере цикл выполняется до тех пор, пока пользователь не введет с клавиатуры символ `A`.

Цикл `for` без тела цикла

Следует учесть, что оператор может быть пустым. Это значит, что тело цикла `for` (или любого другого цикла) также может быть пустым. Такую особенность цикла `for` можно использовать для упрощения некоторых программ, а также в циклах, предназначенных для того, чтобы отложить выполнение последующей части программы на некоторое время.

Программисту иногда приходится решать задачу удаления пробелов из входного потока. Допустим, программа, работающая с базой данных, обрабатывает запрос “показать все балансы меньше 400”. База данных требует представления каждого слова отдельно, без пробелов, т.е. обработчик распознает слово “показать”, но не “показать”. В следующем примере цикл `for` удаляет начальные пробелы в строке `str`:

```
for ( ; *str == ' '; str++) ;
```

В этом примере указатель `str` переставляется на первый символ, не являющийся пробелом. Цикл не имеет тела, так как в нем нет необходимости.¹

Иногда возникает необходимость отложить выполнение последующей части программы на определенное время. Это можно сделать с помощью цикла `for` следующим образом:

```
for(t=0; t < SOME_VALUE; t++) ;
```

Единственное назначение этого цикла — задержка выполнения последующей части программы. Однако следует иметь в виду, что компилятор может оптимизировать объектный код таким образом, что пропустит этот цикл вообще, поскольку он не выполняет никаких действий, тогда желаемой задержки выполнения последующей части программы не произойдет.

Объявление переменных внутри цикла

В стандартах C99 и C++ (но не C89!) допускается объявление переменных в секции инициализации цикла `for`. Объявленная таким образом переменная является локальной переменной цикла и ее область действия распространяется на тело цикла.

Рассмотрим следующий пример:

¹ Этот пример, конечно, учебный. На практике так поступать со строкой не рекомендуется, потому что начало строки `str`, “напрасно висящее” в памяти, впоследствии может создать некоторые трудности. Например, если вы захотите освободить память, занимаемую данной строкой, вам потребуется указать на начало строки, а не на первый отличный от пробела символ в этой строке. — *Прим. перев.*

```
/* Здесь переменная i является локальной
   переменной цикла, а j видима вне цикла.
   *** Этот пример в C89 неправильный! ***/
int j;
for (int i = 0; i<10; i++)
    j = i * i;

/* i = 10;
   Это ошибка, переменная i здесь недоступна! */
```

В данном примере переменная *i* объявлена в секции инициализации цикла `for` и служит параметром цикла. Вне цикла переменная *i* невидима.

Поскольку параметр цикла чаще всего необходим только внутри цикла, его объявление в секции инициализации очень удобно и входит в широкую практику¹. Однако необходимо помнить, что это не поддерживается стандартом C89.

Цикл `while`

Общая форма цикла `while` имеет следующий вид:

```
while (условие) оператор;
```

Здесь *оператор* (тело цикла) может быть пустым оператором, единственным оператором или блоком. *Условие* (управляющее выражение) может быть любым допустимым в языке выражением. *Условие* считается истинным, если значение выражения не равно нулю, а *оператор* выполняется, если условие принимает значение ИСТИНА. Если условие принимает значение ЛОЖЬ, программа выходит из цикла и выполняется следующий за циклом оператор.

В следующем примере ввод с клавиатуры происходит до тех пор, пока пользователь не введет символ `A`:

```
char wait_for_char(void)
{
    char ch;

    ch = '\0'; /* инициализация ch */
    while(ch != 'A') ch = getchar();
    return ch;
}
```

Переменная *ch* является локальной, ее значение при входе в функцию произвольно, поэтому сначала значение *ch* инициализируется нулем. Условие цикла `while` истинно, если *ch* не равно `A`. Поскольку *ch* инициализировано нулем, условие истинно и цикл начинает выполняться. Условие проверяется при каждом нажатии клавиши пользователем. При вводе символа `A` условие становится ложным и выполнение цикла прекращается.

Как и в цикле `for`, в цикле `while` условие проверяется перед началом итерации. Это значит, что если условие ложно, тело цикла не будет выполнено. Благодаря этому нет необходимости вводить в программу отдельное условие перед циклом. Рассмотрим это на примере функции `pad()`, которая добавляет пробелы в конец строки и делает ее длину равной предварительно заданной величине. Если строка уже имеет необходимую длину, то пробелы не добавляются:

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

void pad(char *s, int length);
```

¹ В некоторых языках (например АЛГОЛ 68) локализация параметра цикла выполняется автоматически. — *Прим. ред.*

```

int main(void)
{
    char str[80];

    strcpy(str, "это проверка");
    pad(str, 40);
    printf("%d", strlen(str));

    return 0;
}

/* Добавление пробелов в конец строки. */
void pad(char *s, int length)
{
    int l;

    l = strlen(s); /* определение длины строки */

    while(l < length) {
        s[l] = ' '; /* вставка пробела */
        l++;
    }
    s[l] = '\0'; /* строка должна заканчиваться нулем */
}

```

Аргументами функции `pad()` являются `s` (указатель на исходную строку) и `length` (требуемое количество символов в строке). Если длина строки `s` при входе в функцию равна или больше `length`, то цикл `while` не выполняется. В противном случае `pad()` добавляет требуемое количество пробелов, а библиотечная функция `strlen()` возвращает длину строки.

Если выполнение цикла должно зависеть от нескольких условий, можно создать так называемую управляющую переменную, значения которой присваиваются разными операторами тела цикла. Рассмотрим следующий пример:

```

void func1(void)
{
    int working;

    working = 1; /* то есть ИСТИНА */

    while(working) {
        working = process1();
        if(working)
            working = process2();
        if(working)
            working = process3();
    }
}

```

В этом примере переменная `working` является управляющей. Любая из трех функций может вернуть значение 0 и этим прервать выполнение цикла.

Тело цикла `while` может быть пустым. Например, цикл

```
while((ch=getchar()) != 'A') ;
```

выполняется до тех пор, пока пользователь не введет символ 'A'. Напоминаем, что оператор присваивания выполняет две задачи: присваивает значение выражения справа переменной слева и возвращает это значение как свое собственное.

Цикл do-while

В отличие от циклов `for` и `while`, которые проверяют свое условие перед итерацией, `do-while` делает это после нее. Поэтому цикл `do-while` всегда выполняется как минимум один раз. Общая форма цикла `do-while` следующая:

```
do {  
    оператор;  
} while (условие);
```

Если *оператор* не является блоком, фигурные скобки не обязательны, но их почти всегда ставят, чтобы оператор достаточно наглядно отделялся от условия. Итерации оператора `do-while` выполняются, пока *условие* не примет значение ЛОЖЬ.

В следующем примере в цикле `do-while` числа считываются с клавиатуры, пока не встретится число, меньшее или равное 100:

```
do {  
    scanf("%d", &num);  
} while(num > 100);
```

Цикл `do-while` часто используется в функциях выбора пунктов меню. Если пользователь вводит допустимое значение, оно возвращается в качестве значения функции. В противном случае цикл требует повторить ввод. Следующий пример демонстрирует усовершенствованную версию программы для выбора пункта меню проверки грамматики:

```
void menu(void)  
{  
    char ch;  
  
    printf("1. Проверка правописания\n");  
    printf("2. Коррекция ошибок\n");  
    printf("3. Вывод ошибок\n");  
    printf("    Введите ваш выбор: ");  
    do {  
        ch = getchar(); /* чтение выбора с клавиатуры */  
  
        switch(ch) {  
            case '1':  
                check_spelling();  
                break;  
            case '2':  
                correct_errors();  
                break;  
            case '3':  
                display_errors();  
                break;  
        }  
    } while(ch!='1' && ch!='2' && ch!='3');  
}
```

В этом примере применение цикла `do-while` весьма уместно, потому что итерация, как уже упоминалось, всегда должна выполняться как минимум один раз. Цикл повторяется, пока его условие не станет ложным, т.е. пока пользователь не введет один из допустимых ответов.

Операторы перехода

В языке C определены четыре оператора перехода: `return`, `goto`, `break` и `continue`. Операторы `return` и `goto` можно использовать в любом месте внутри функции. Операторы `break` и `continue` можно использовать в любом из операторов цикла. Как указывалось ранее в этой главе, `break` можно также использовать в операторе `switch`.

Оператор `return`

Оператор `return` используется для выхода из функции. Отнесение его к категории операторов перехода обусловлено тем, что он заставляет программу перейти в точку вызова функции. Оператор `return` может иметь ассоциированное с ним значение, тогда при выполнении данного оператора это значение возвращается в качестве значения функции. В функциях типа `void` используется оператор `return` без значения.

Стандарт C89 допускает наличие оператора `return` без значения, даже если тип функции отличен от `void`. В этом случае функция возвращает неопределенное значение. Но что касается языков C99 и C++, если тип функции отличен от `void`, то ее оператор `return` обязательно должен иметь значение. Конечно, и в программе на C89 отсутствие возвращаемого значения в функции, тип которой отличен от `void`, является признаком плохого стиля!

Общая форма оператора `return` следующая:

`return выражение;`

Выражение присутствует только в том случае, если функция возвращает значение. Это значение *выражения* становится возвращаемым значением функции.

Внутри функции может присутствовать произвольное количество операторов `return`. Выход из функции происходит тогда, когда встречается один из них. Закрывающаяся фигурная скобка `}` также вызывает выход из функции. Выход программы на нее эквивалентен оператору `return` без значения. В этом случае функция, тип которой отличен от `void`, возвращает неопределенное значение.

Функция, определенная со спецификатором `void`, не может содержать `return` со значением. Так как эта функция не возвращает значения, в ней не может быть оператора `return`, возвращающего значение. Более подробно `return` рассматривается в главе 6.

Оператор `goto`

Кроме `goto`, в языке C есть другие операторы управления (например `break`, `continue`), поэтому необходимости в применении `goto` практически нет. В результате чрезмерного использования операторов `goto` программа плохо читается, она становится “похожей на спагетти”. Чаще всего такими программами недовольна администрация фирм, производящих программный продукт. То есть оператор `goto` весьма непопулярен, более того, считается, что в программировании не существует ситуаций, в которых нельзя обойтись без оператора `goto`. Но в некоторых случаях его применение все же уместно. Иногда, при умелом использовании, этот оператор может оказаться весьма полезным, например, если нужно покинуть глубоко вложенные циклы¹. В данной книге оператор `goto` рассматривается только в этом разделе.

¹ Уже одно это (чрезмерная вложенность и неожиданный выход сразу из нескольких циклов) может свидетельствовать о плохой структуре программы. — *Прим. ред.*

Для оператора `goto` всегда необходима метка. *Метка* — это идентификатор с последующим двоеточием. Метка должна находиться в той же функции, что и `goto`, переход в другую функцию невозможен. Общая форма оператора `goto` следующая:

```
goto метка;
```

```
.  
.   
.
```

```
метка:
```

Метка может находиться как до, так и после оператора `goto`. Например, используя оператор `goto`, можно выполнить цикл от 1 до 100:

```
x = 1;  
loop1:  
    x++;  
if(x <= 100) goto loop1;
```

Оператор `break`

Оператор `break` применяется в двух случаях. Во-первых, в операторе `switch` с его помощью прерывается выполнение последовательности `case` (см. раздел “Оператор выбора — `switch`” ранее в этой главе). В этом случае оператор `break` не передает управление за пределы блока. Во-вторых, оператор `break` используется для немедленного прекращения выполнения цикла без проверки его условия, в этом случае оператор `break` передает управление оператору, следующему после оператора цикла.

Когда внутри цикла встречается оператор `break`, выполнение цикла безусловно (т.е. без проверки каких-либо условий. — *Прим. ред.*) прекращается и управление передается оператору, следующему за ним. Например, программа

```
#include <stdio.h>  
  
int main(void)  
{  
    int t;  
  
    for(t=0; t < 100; t++) {  
        printf("%d ", t);  
        if(t == 10) break;  
    }  
  
    return 0;  
}
```

выводит на экран числа от 0 до 10. После этого выполнение цикла прекращается оператором `break`, условие `t < 100` при этом игнорируется.

Оператор `break` часто используется в циклах, в которых некоторое событие должно вызвать немедленное прекращение выполнения цикла. В следующем примере нажатие клавиши прекращает выполнение функции `look_up()`:

```
void look_up(char *name)  
{  
    do {  
        /* поиск имени 'name' */  
        if(kbhit()) break;  
    } while(!found);  
}
```

Библиотечная функция `kbhit()` возвращает 0, если клавиша не нажата (то есть, буфер клавиатуры пуст), в противном случае она возвращает ненулевое значение. В стандарте C функция `kbhit()` не определена, однако практически она поставляется почти с каждым компилятором (возможно, под несколько другим именем).

Оператор `break` вызывает выход только из внутреннего цикла. Например, программа

```
for(t=0; t < 100; ++t) {
    count = 1;
    for(;;) {
        printf("%d ", count);
        count++;
        if(count == 10) break;
    }
}
```

100 раз выводит на экран числа от 1 до 9. Оператор `break` передает управление внешнему циклу `for`.

Если оператор `break` присутствует внутри оператора `switch`, который вложен в какие-либо циклы, то `break` относится только к `switch`, выход из цикла не происходит.

Функция `exit()`

Функция `exit()` не является оператором языка, однако рассмотрим возможность ее применения. Аналогично прекращению выполнения цикла оператором `break`, можно прекратить работу программы и с помощью вызова стандартной библиотечной функции `exit()`. Эта функция вызывает немедленное прекращение работы всей программы и передает управление операционной системе.

Общая форма функции `exit()` следующая:

```
void exit (int код_возврата);
```

Значение переменной *код_возврата* передается вызвавшему программу процессу, обычно в качестве этого процесса выступает операционная система. Нулевое значение кода возврата обычно используется для указания нормального завершения работы программы. Другие значения указывают на характер ошибки. В качестве кода возврата можно использовать макросы `EXIT_SUCCESS` и `EXIT_FAILURE` (выход успешный и выход с ошибкой). Функция `exit()` объявлена в заголовочном файле `<stdlib.h>`.

Функция `exit()` часто используется, когда обязательное условие работы программы не выполняется. Рассмотрим, например, компьютерную игру в виртуальной реальности, использующую специальный графический адаптер. Главная функция `main()` этой игры выглядит так:

```
#include <stdlib.h>

int main(void)
{
    if(!virtual_graphics()) exit(1);
    play();
    /* */
}
```

Здесь `virtual_graphics()` возвращает значение ИСТИНА, если присутствует нужный графический адаптер. Если требуемого адаптера нет, вызов функции `exit(1)` прекращает работу программы.

В следующем примере в новой версии ранее рассмотренной функции `menu()` вызов `exit()` используется для выхода из программы и возврата в операционную систему:

```

void menu(void)
{
    char ch;

    printf("1. Проверка правописания\n");
    printf("2. Коррекция ошибок\n");
    printf("3. Вывод ошибок\n");
    printf("4. Выход из программы\n");
    printf("    Введите ваш выбор: ");
    do {
        ch = getchar(); /* чтение клавиши */

        switch(ch) {
            case '1':
                check_spelling();
                break;
            case '2':
                correct_errors();
                break;
            case '3':
                display_errors();
                break;
            case '4':
                exit(0); /* Возвращение в ОС */
        }
    } while(ch!='1' && ch!='2' && ch!='3');
}

```

Оператор continue

Можно сказать, что оператор `continue` немного похож на `break`. Оператор `break` вызывает прерывание цикла, а `continue` — прерывание текущей итерации цикла и осуществляет переход к следующей итерации. При этом все операторы до конца тела цикла пропускаются. В цикле `for` оператор `continue` вызывает выполнение операторов приращения и проверки условия цикла. В циклах `while` и `do-while` оператор `continue` передает управление операторам проверки условий цикла. В следующем примере программа подсчитывает количество пробелов в строке, введенной пользователем:

```

/* Подсчет количества пробелов */
#include <stdio.h>

int main(void)
{
    char s[80], *str;
    int space;

    printf(" Введите строку: ");
    gets(s);
    str = s;

    for(space=0; *str; str++) {
        if(*str != ' ') continue;
        space++;
    }
    printf("%d пробелов\n", space);

    return 0;
}

```

Каждый символ строки сравнивается с пробелом. Если сравниваемый символ не является пробелом, оператор `continue` передает управление в конец цикла `for` и выполняется следующая итерация. Если символ является пробелом, значение переменной `space` увеличивается на 1.

В следующем примере оператор `continue` применяется для выхода из цикла `while` путем передачи управления на условие цикла:

```
void code(void)
{
    char done, ch;

    done = 0;
    while(!done) {
        ch = getchar();
        if(ch == '$') {
            done = 1;
            continue;
        }
        putchar(ch+1); /* печать следующего в алфавитном порядке
                        символа */
    }
}
```

Функция `code` предназначена для кодирования сообщения путем замены каждого символа символом, код которого на 1 больше кода исходного символа в коде ASCII. Например, символ A заменяется символом B (если это латинские символы. — *Прим. перев.*). Функция прекращает работу при вводе символа \$. При этом переменной `done` присваивается значение 1 и оператор `continue` передает управление на условие цикла, что и прекращает выполнение цикла.

Оператор-выражение

Выражения были подробно рассмотрены в главе 2. Но говоря об операторах, будет уместно добавить несколько слов и о выражениях. Любое выражение, которое заканчивается точкой с запятой, является оператором. Рассмотрим следующие примеры:

```
func(); /* вызов функции */
a = b+c; /* оператор присваивания */
b+f(); /* правильный, но "странный" оператор */
; /* пустой оператор */
```

Первый оператор выполняет вызов функции, второй — присваивание. Третий оператор выглядит странно, но транслятор все же не укажет на ошибку (возможно, даст предупреждение). В этом операторе необходимые действия, видимо, выполняются функцией `f()`. Последний пример — *пустой оператор*, не выполняющий никакого действия.

Блок операторов

Блок — это последовательность операторов, заключенных в фигурные скобки и рассматриваются как одна программная единица. Операторы, составляющие блок, логически связаны друг с другом. Иногда блок называют *составным оператором*. Блок всегда начинается открывающейся фигурной скобкой `{` и заканчивается закрывающейся `}`. Чаще всего блок используется как составная часть какого-либо оператора,

выполняющего действие над группой операторов, например, `if` или `for`. Однако блок можно поставить в любом месте, где может находиться оператор, как это показано в следующем примере:

```
#include <stdio.h>

int main(void)
{
    int i;

    {          /* блок операторов */
        i = 120;
        printf("%d", i);
    }

    return 0;
}
```

Полный
справочник по



Глава 4

Массивы и строки

Массив — это набор переменных одного типа, имеющих одно и то же имя. Доступ к конкретному элементу массива осуществляется с помощью индекса. В языке С все массивы располагаются в отдельной непрерывной области памяти. Первый элемент массива располагается по самому меньшему адресу, а последний — по самому большому. Массивы могут быть одномерными и многомерными. *Строка* представляет собой массив символьных переменных, заканчивающийся специальным нулевым символом, это наиболее распространенный тип массива.

Массивы и указатели тесно связаны. То, что может быть сказано о массивах, чаще всего непосредственно относится и к указателям, и наоборот. В этой главе рассматриваются массивы, указатели будут подробно рассмотрены в главе 5. Для полного понимания приемов работы с массивами и указателями читателю следует хорошо усвоить обе эти главы.

Одномерные массивы

Общая форма объявления одномерного массива имеет следующий вид:

тип имя_переменной [размер];

Как и другие переменные, массив должен быть объявлен явно, чтобы компилятор выделил для него определенную область памяти (т.е. разместил массив). Здесь *тип* обозначает базовый тип массива, являющийся типом каждого элемента. *Размер* задает количество элементов массива. Например, следующий оператор объявляет массив из 100 элементов типа `double` под именем `balance`:

```
double balance[100];
```

Согласно стандарту С89 размер массива должен быть указан явно с помощью выражения-константы. Таким образом, в программе на С89 размер массива определяется во время компиляции и впоследствии остается неизменным. (В С99 определены массивы, размер которых определяется во время выполнения. О них еще будет идти речь далее в этой главе, а также более подробно в части II).

Доступ к элементу массива осуществляется с помощью имени массива и индекса. Индекс элемента массива помещается в квадратные скобки после имени. Например, оператор

```
balance[3] = 12.23;
```

присваивает 3-му элементу массива `balance` значение 12.23.

Индекс первого элемента любого массива в языке С равен нулю. Поэтому оператор

```
char p[10];
```

объявляет массив символов из 10 элементов — от `p[0]` до `p[9]`. В следующей программе вычисляются значения элементов массива целого типа с индексами от 0 до 99:

```
#include <stdio.h>

int main(void)
{
    int x[100]; /* объявление массива 100 целых */
    int t;

    /* присваивание массиву значений от 0 до 99 */
    for(t=0; t<100; t++) x[t] = t;

    /* вывод на экран содержимого x */
```

```

    for(t=0; t<100; t++) printf("%d ", x[t]);

    return 0;
}

```

Объем памяти, необходимый для хранения массива, непосредственно определяется его типом и размером. Для одномерного массива количество байтов памяти вычисляется следующим образом:

количество_байтов = sizeof(базовый_тип) × длина_массива

Во время выполнения программы на С не проверяется ни соблюдение границ массивов, ни их содержимое. В область памяти, занятую массивом, может быть записано что угодно, даже программный код. Программист должен сам, где это необходимо, ввести проверку границ индексов. Следующий пример программы компилируется без ошибки, однако при выполнении происходит нарушение границы массива `count` и разрушение соседних участков памяти:

```

int count[10], i;

/* здесь нарушена граница массива count */
for(i=0; i<100; i++ count[i] = i;

```

Можно сказать, что одномерный массив — это список, хранящийся в непрерывной области памяти в порядке индексации. На рис. 4.1 показано, как хранится в памяти массив `a`, начинающийся по адресу 1000 и объявленный как

```
char a[7];
```

Элемент	a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]
Адрес	1000	1001	1002	1003	1004	1005	1006

Рис. 4.1. Массив из семи символов, начинающийся по адресу 1000



Создание указателя на массив

Указатель на 1-й элемент массива можно создать путем присваивания ему имени массива без индекса. Например, если есть объявление

```
int sample[10];
```

то в качестве указателя на 1-й элемент массива можно использовать имя `sample`. В следующем фрагменте программы адрес 1-го элемента массива `sample` присваивается указателю `p`:

```

int *p;
int sample[10];

p = sample;

```

В обеих переменных (`p` и `sample`) хранится адрес 1-го элемента, отличаются эти переменные только тем, что значение `sample` в программе изменять нельзя. Адрес первого элемента можно также получить, используя оператор получения адреса `&`. Например, выражения `sample` и `&sample[0]` имеют одно и то же значение. Тем не менее, в профессионально написанных программах вы не встретите выражения `&sample[0]`.



Передача одномерного массива в функцию

В языке С нельзя передать весь массив как аргумент функции. Однако можно передать указатель на массив, т.е. имя массива без индекса. Например, в представленной программе в `func1()` передается указатель на массив `i`:

```
int main(void)
{
    int i[10];

    func1(i);

    /* */
}
```

Если в функцию передается указатель на одномерный массив, то в самой функции его можно объявить одним из трех вариантов: как указатель, как массив определенного размера и как массив без определенного размера. Например, чтобы функция `func1()` получила доступ к значениям, хранящимся в массиве `i`, она может быть объявлена как

```
void func1(int *x) /* указатель */
{
    /* */
}
```

или как

```
void func1(int x[10]) /* массив определенного размера */
{
    /* */
}
```

и наконец как

```
void func1(int x[]) /* массив без определенного размера */
{
    /* */
}
```

Эти три объявления тождественны, потому что каждое из них сообщает компилятору одно и то же: в функцию будет передан указатель на переменную целого типа. В первом объявлении используется указатель, во втором — стандартное объявление массива. В последнем примере измененная форма объявления массива сообщает компилятору, что в функцию будет передан массив неопределенной длины. Как видно, длина массива не имеет для функции никакого значения, потому что в С проверка границ массива не выполняется. Эту функцию можно объявить даже так:

```
void func1(int x[32])
{
    /* ... */
}
```

И при этом программа будет выполнена правильно, потому что компилятор не создает массив из 32 элементов, а только подготавливает функцию к приему указателя.



Строки

Одномерный массив наиболее часто применяется в виде строки символов. *Строка* — это одномерный массив символов, заканчивающийся нулевым символом. В языке С признаком окончания строки (нулевым символом) служит символ `'\0'`. Таким

образом, строка содержит символы, составляющие строку, а также нулевой символ. Это единственный вид строки, определенный в С.

На заметку

В С++ дополнительно определен специальный класс строк, называющийся *String*¹, который позволяет обрабатывать строки объектно-ориентированными методами. Стандарт С не поддерживает *String*.

Объявляя массив символов, предназначенный для хранения строки, необходимо предусмотреть место для нуля, т.е. указать его размер в объявлении на один символ больше, чем наибольшее предполагаемое количество символов. Например, объявление массива *str*, предназначенного для хранения строки из 10 символов, должно выглядеть так:

```
char str[11];
```

Последний, 11-й байт предназначен для нулевого символа.

Записанная в тексте программы строка символов, заключенных в двойные кавычки, является *строковой константой*, например, "некоторая строка"

В конец строковой константы компилятор автоматически добавляет нулевой символ.

Для обработки строк в С определено много различных библиотечных функций. Чаще всего используются следующие функции:

Имя функции	Выполняемое действие
<code>strcpy(s1,s2)</code>	Копирование <i>s2</i> в <i>s1</i>
<code>strcat(s1,s2)</code>	Конкатенация (присоединение) <i>s2</i> в конец <i>s1</i>
<code>strlen(s1)</code>	Возвращает длину строки <i>s1</i>
<code>strcmp(s1,s2)</code>	Возвращает 0, если <i>s1</i> и <i>s2</i> совпадают, отрицательное значение, если <i>s1</i> < <i>s2</i> и положительное значение, если <i>s1</i> > <i>s2</i>
<code>strchr(s1,ch)</code>	Возвращает указатель на первое вхождение символа <i>ch</i> в строку <i>s1</i>
<code>strstr(s1,s2)</code>	Возвращает указатель на первое вхождение строки <i>s2</i> в строку <i>s1</i>

Эти функции объявлены в заголовочном файле `<string.h>`. Применение библиотечных функций обработки строк иллюстрируется следующим примером:

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

int main(void)
{
    char s1[80], s2[80];

    gets(s1);
    gets(s2);

    printf("Длина: %d %d\n", strlen(s1), strlen(s2));

    if(!strcmp(s1, s2)) printf("Строки равны\n");

    strcat(s1, s2);
    printf("%s\n", s1);

    strcpy(s1, "Проверка.\n");
    printf(s1);
    if(strchr("Алло!", 'л')) printf(" л есть в Алло\n");
    if(strstr("Привет!", "ив")) printf(" найдено ив ");
}
```

¹ CString в среде Visual C++. — Прим. перев.

```
    return 0;
}
```

Если эту программу выполнить и ввести в `s1` и в `s2` одну и ту же строку “Алло!”, то на экран будет выведено следующее:

```
Длина: 5 5
Строки равны
Алло!Алло!
Проверка.
л есть в Алло
найденo ив
```

Следует помнить, что `strcmp()` принимает значение ЛОЖЬ, если строки совпадают (хоть это и несколько нелогично). Поэтому в тесте на совпадение нужно использовать логический оператор отрицания `!` как в предыдущем примере.

Двухмерные массивы

Стандартом C определены многомерные массивы. Простейшая форма многомерного массива — двухмерный массив. Двухмерный массив — это массив одномерных массивов. Объявление двухмерного массива `d` с размерами 10 на 20 выглядит следующим образом:

```
int d[10][20];
```

Во многих языках измерения массива отделяются друг от друга запятой. В языке C каждое измерение заключено в свои квадратные скобки.

Аналогично обращению к элементу одномерного массива, обращение к элементу с индексами 1 и 2 двухмерного массива `d` выглядит так:

```
d[1][2]
```

В следующем примере элементам двухмерного массива присваиваются числа от 1 до 12 и значения элементов выводятся на экран построчно:

```
#include <stdio.h>

int main(void)
{
    int t, i, num[3][4];

    for(t=0; t<3; t++)
        for(i=0; i<4; i++)
            num[t][i] = (t*4)+i+1;

    /* вывод на экран */
    for(t=0; t<3; t++) {
        for(i=0; i<4; i++)
            printf("%3d ", num[t][i]);
        printf("\n");
    }

    return 0;
}
```

В этом примере `num[0][0]` имеет значение 1, `num[0][1]` — значение 2, `num[0][2]` — значение 3 и так далее. Наглядно двухмерный массив `num` можно представить так:

```

num[t][i]
      0   1   2   3
0  1   2   3   4
2  5   6   7   8
3  9  10  11  12

```

Двухмерные массивы размещаются в матрице, состоящей из строк и столбцов. Первый индекс указывает номер строки, а второй — номер столбца. Это значит, что когда к элементам массива обращаются в том порядке, в котором они размещены в памяти, правый индекс изменяется быстрее, чем левый. На рис. 4.2 показано графическое представление двухмерного массива в памяти.

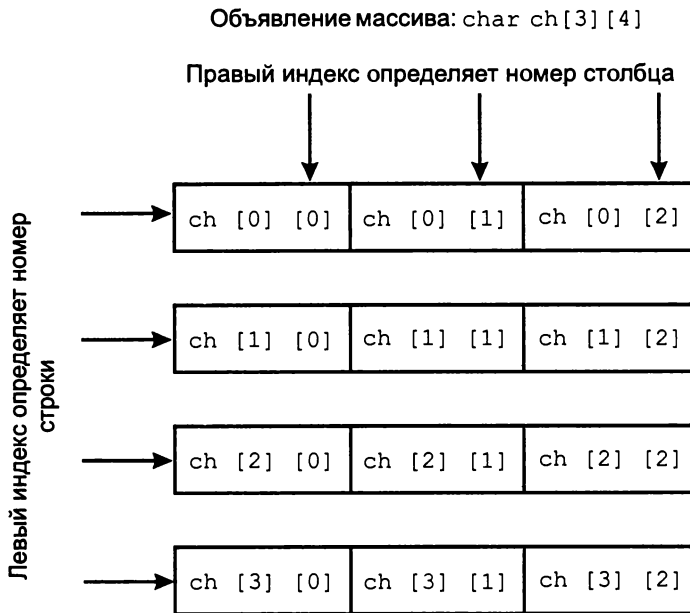


Рис. 4.2. Двухмерный массив

Объем памяти в байтах, занимаемый двухмерным массивом, вычисляется по следующей формуле:

количество_байтов =
= размер_1-го_измерения × размер_2-го_измерения × sizeof(базовый_тип)

Например, двухмерный массив 4-байтовых целых чисел размерностью 10×5 занимает участок памяти объемом

$10 \times 5 \times 4$

то есть 200 байтов.

Если двухмерный массив используется в качестве аргумента функции, то в нее передается только указатель на начальный элемент массива. В соответствующем параметре функции, получающем двухмерный массив, обязательно должен быть указан размер правого измерения¹, который равен длине строки массива. Размер левого измерения указывать не обязательно. Размер правого измерения необходим компилятору для того, чтобы

¹ Размер правого измерения указывать не нужно, если в вызывающей функции массив объявлен как `**x` и размещен динамически (см. главу 5). — Прим. перев.

внутри функции правильно вычислить адрес элемента массива, так как для этого компилятор должен знать длину строки массива. Например, функция, получающая двухмерный массив целых размерностью 10×10, должна быть объявлена так:

```
void func1(int x[][10])
{
    /* ... */
}
```

Компилятор должен знать длину строки массива, чтобы внутри функции правильно вычислить адрес элемента массива. Если при компиляции функции это неизвестно, то невозможно определить, где начинается следующая строка, и вычислить, например, адрес элемента

```
x[2][4]
```

В следующем примере двумерные массивы используются для хранения оценок студентов. Предполагается, что преподаватель ведет три класса, в каждом из которых учится не более 30 студентов. Обратите внимание на то, как происходит обращение к массиву `grade` в каждой функции.

```
/* Простая база данных оценок студентов */
#include <stdio.h>
#include <ctype.h>
#include <stdlib.h>

#define CLASSES 3
#define GRADES 30

int grade[CLASSES][GRADES];

void enter_grades(void);
int get_grade(int num);
void disp_grades(int g[][GRADES]);

int main(void)
{
    char ch, str[80];

    for(;;) {
        do {
            printf("(В)вод оценок студентов\n");
            printf("В(ы)вод оценок студентов\n");
            printf("Вы(х)од из программы\n");
            gets(str);
            ch = toupper(*str);
        } while(ch!='B' && ch!='ы' && ch!='х');

        switch(ch) {
            case 'B':
                enter_grades();
                break;
            case 'ы':
                disp_grades(grade);
                break;
            case 'х':
                exit(0);
        }
    }
}
```

```

    return 0;
}

/* Занесение оценок студентов в массив */
void enter_grades(void)
{
    int t, i;

    for(t=0; t<CLASSES; t++) {
        printf("Класс № %d:\n", t+1);
        for(i=0; i<GRADES; i++)
            grade[t][i] = get_grade(i);
    }
}

/* Ввод оценок */
int get_grade(int num)
{
    char s[80];

    printf("Введите оценку студента № %d:\n", num+1);
    gets(s);
    return(atoi(s));
}

/* Вывод оценок */
void disp_grades(int g[][GRADES])
{
    int t, i;

    for(t=0; t<CLASSES; t++) {
        printf("Класс № %d:\n", t+1);
        for(i=0; i<GRADES; i++)
            printf("Студент № %d имеет оценку %d\n",
                    i+1, g[t][i]);
    }
}

```

Массивы строк

В программах на языке С часто используются массивы строк. Например, сервер базы данных сверяет команды пользователей с массивом допустимых команд. В качестве массива строк в языке С служит двухмерный символьный массив. Размер левого измерения определяет количество строк, а правого — максимальную длину каждой строки. Например, в следующем операторе объявлен массив из 30 строк с максимальной длиной 79 символов:

```
■ char str_array[30][80];
```

Чтобы обратиться к отдельной строке массива, нужно указать только левый индекс. Например, вызов функции `gets()` с третьей строкой массива `str_array` в качестве аргумента можно записать так:

```
■ gets(str_array[2]);
```

Этот оператор эквивалентен следующему:

```
■ gets(&str_array[2][0]);
```

Из этих двух форм записи предпочтительной является первая.

Для лучшего понимания свойств массива строк рассмотрим следующую короткую программу, в которой на основе применения массива строк создан простой текстовый редактор:

```
/* Очень простой текстовый редактор */
#include <stdio.h>

#define MAX 100
#define LEN 80

char text[MAX][LEN];

int main(void)
{
    register int t, i, j;

    printf("Для выхода введите пустую строку.\n");

    for(t=0; t<MAX; t++) {
        printf("%d: ", t);
        gets(text[t]);
        if(!*text[t]) break; /* выход по пустой строке */
    }

    for(i=0; i<t; i++) {
        for(j=0; text[i][j]; j++) putchar(text[i][j]);
        putchar('\n');
    }

    return 0;
}
```

Пользователь вводит в программу строки текста, заканчивая ввод пустой строкой. Затем программа выводит текст посимвольно.



Многомерные массивы

В языке С можно пользоваться массивами, размерность которых больше двух. Общая форма объявления многомерного массива следующая:

тип имя_массива [Размер1][Размер2]...[РазмерN];

Массивы, у которых число измерений больше трех, используются довольно редко, потому что они занимают большой объем памяти. Например, четырехмерный массив символов размерностью $10 \times 6 \times 9 \times 4$ занимает 2160 байтов. Если бы массив содержал 2-байтовые целые, потребовалось бы 4320 байтов. Если бы элементы массива имели тип `double`, причем каждый элемент (вещественное число двойной точности) занимал бы 8 байтов, то для хранения массива потребовалось бы 17 280 байтов. Объем требуемой памяти с ростом числа измерений растет экспоненциально. Например, если к предыдущему массиву добавить пятое измерение, причем его толщину по этому измерению сделать равной всего 10, то его объем возрастет до 172 800 байтов.

При обращении к многомерным массивам компьютер много времени затрачивает на вычисление адреса, так как при этом приходится учитывать значение каждого индекса. Поэтому доступ к элементам многомерного массива происходит значительно медленнее, чем к элементам одномерного.

Передавая многомерный массив в функцию, в объявлении параметров функции необходимо указать все размеры измерений, кроме самого левого. Например, если массив `m` объявлен как

```
int m[4][3][6][5];
```

то функция, принимающая этот массив, должна быть объявлена примерно так:

```
void func1(int d[][3][6][5])
{
    /* ... */
}
```

Конечно, можно включить в объявление и размер 1-го измерения, но это излишне.

Индексация указателей

Указатели и массивы тесно связаны друг с другом. Имя массива без индекса — это указатель на первый (начальный) элемент массива. Рассмотрим, например, следующий массив:

```
char p[10];
```

Следующие два выражения идентичны:

```
p
&p[0]
```

Выражение

```
p == &p[0]
```

принимает значение **ИСТИНА**, потому что адрес 1-го элемента массива — это то же самое, что и адрес массива.

Как уже указывалось, имя массива без индекса представляет собой указатель. И наоборот, указатель можно индексировать как массив. Рассмотрим следующий фрагмент программы:

```
int *p, i[10];
p = i;
p[5] = 100; /* в присваивании используется индекс */
*(p+5) = 100; /* в присваивании используется адресная арифметика */
```

Оба оператора присваивания заносят число 100 в 6-й элемент массива `i`. Первый из них индексирует указатель `p`, во втором применяются правила адресной арифметики. В обоих случаях получается один и тот же результат. (Подробно указатели и адресная арифметика рассматриваются в главе 5.)

Можно также индексировать указатели на многомерные массивы. Например, если `a` — это указатель на двухмерный массив целых размерностью 10×10 , то следующие два выражения эквивалентны:

```
a
&a[0][0]
```

Более того, к элементу $(0, 4)^1$ можно обратиться двумя способами: либо указав индексы массива: `a[0][4]`, либо с помощью указателя: `*((int*)a+4)`. Аналогично для элемента $(1, 2)$: `a[1][2]` или `*((int*)a+12)`. В общем виде для двухмерного массива справедлива следующая формула:

`a[j][k]` эквивалентно `*((базовый_тип*)a+(j*длина_строки)+k)`

¹ Так обозначается элемент, у которого первая координата равна 0, а вторая — 4. — Прим. ред.

Правила адресной арифметики требуют явного преобразования указателя на массив в указатель на базовый тип (см. главу 5). Указатели используются для обращения к элементам массива потому, что часто операции адресной арифметики выполняются быстрее, чем индексация массива.

Двухмерный массив может быть представлен как указатель на массив одномерных массивов. Добавив еще один указатель, можно с его помощью обращаться к элементам отдельной строки массива. Этот прием демонстрируется в функции `pr_row()`, которая печатает содержимое конкретной строки двухмерного глобального массива `num`:

```
int num[10][10];

/* ... */

void pr_row(int j)
{
    int *p, t;

    p = (int *) &num[j][0]; /* вычисление адреса 1-го
                               элемента строки номер j */

    for(t=0; t<10; t++) printf("%d ", *(p+t));
}
```

Эту функцию можно обобщить, включив в список аргументов номер строки, длину строки и указатель на 1-й элемент:

```
void row(int j, int row_dimension, int *p)
{
    int t;

    p = p + (j * row_dimension);

    for(t=0; t<row_dimension; t++)
        printf("%d ", *(p+t));
}

/* ... */

void f(void)
{
    int num[10][10];

    pr_row(0, 10, (int *) num); /* печать 1-й строки */
}
```

Такой прием “понижения размерности” годится не только для двухмерных массивов, но и для любых многомерных. Например, вместо того, чтобы работать с трехмерным массивом, можно использовать указатель на двухмерный массив, причем вместо него в свою очередь можно использовать указатель на одномерный массив. В общем случае вместо того, чтобы обращаться к n -мерному массиву, можно работать с указателем на $(n-1)$ -мерный массив. Причем этот процесс понижения размерности кончается на одномерном массиве.



Инициализация массивов

В языке C массивы при объявлении можно инициализировать. Общая форма инициализации массива аналогична инициализации переменной:

тип имя_массива[размер1]...[размерN] = {список_значений};

`Список_значений` представляет собой список констант, разделенных запятыми. Типы констант должны быть совместимыми с *типом* массива. Первая константа присваивается первому элементу массива, вторая — второму и так далее. После закрывающейся фигурной скобки точка с запятой обязательна.

На заметку

В С99 локальные массивы можно инициализировать не константами, а переменными, однако в С89 все массивы инициализируются только константами.

В следующем примере массив целых из 10 элементов инициализируется числами от 1 до 10:

```
int i[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
```

Здесь элементу `i[0]` присваивается 1, а `i[9]` — 10.

Символьные массивы, содержащие строки, можно инициализировать строковыми константами:

```
char имя_массива[размер] = "строка";
```

В следующем примере массив `str` инициализируется фразой "Язык С":

```
char str[7] = "Язык С";
```

Это объявление можно записать так:

```
char str[7] = {'Я', 'з', 'ы', 'к', ' ', 'С', '\0'};
```

Строка кончается нулевым символом, поэтому при объявлении необходимо задавать размер массива, достаточный для того, чтобы этот символ поместился в нем. В предыдущем примере размер строки задан равным 7, хотя во фразе "Язык С" содержится 6 символов. Если строка инициализируется строковой константой, компилятор автоматически добавляет нулевой символ в конец строки.

Многомерные массивы инициализируются так же, как и одномерные. В следующем примере массив `sqr` инициализируется числами от 1 до 10 и их квадратами:

```
int sqr[10][2] = {
    1, 1,
    2, 4,
    3, 9,
    4, 16,
    5, 25,
    6, 36,
    7, 49,
    8, 64,
    9, 81,
    10, 100
};
```

Инициализируя многомерный массив, для улучшения наглядности элементы инициализации каждого измерения можно заключать в фигурные скобки. Этот способ называется *группированием подагрегатов* (*subaggregate grouping*). С использованием этого приема предыдущий пример может быть записан так:

```
int sqr[10][2] = {
    {1, 1},
    {2, 4},
    {3, 9},
    {4, 16},
    {5, 25},
    {6, 36},
    {7, 49},
    {8, 64},
    {9, 81},
    {10, 100}
};
```

```
{9, 81},
{10, 100}
};
```

При такой записи, если внутри группы недостаточно констант инициализации, то оставшиеся элементы группы автоматически заполняются нулями.

Инициализация безразмерных массивов

Предположим, что необходимо создать таблицу сообщений об ошибках, используя инициализацию массивов:

```
char e1[15] = "Ошибка чтения\n";
char e2[15] = "Ошибка записи\n";
char e3[21] = "Нельзя открыть файл\n";
```

Для задания размера массива пришлось бы вручную подсчитывать количество символов в каждом сообщении. Однако в языке С есть конструкция, благодаря которой компилятор автоматически определяет необходимую длину строки. Если в операторе инициализации массива не указан размер массива, компилятор создает массив такого размера, что в нем умещаются все инициализирующие элементы. Таким образом создается *безразмерный массив*. Используя этот метод, предыдущий пример можно записать так:

```
char e1[] = "Ошибка чтения\n";
char e2[] = "Ошибка записи\n";
char e3[] = "Нельзя открыть файл\n";
```

Тогда оператор

```
printf("%s имеет длину %d\n", e2, sizeof e2);
```

выведет на экран следующее:

```
Ошибка записи
имеет длину 15
```

Кроме уменьшения трудоемкости, инициализация безразмерных массивов полезна тем, что позволяет изменять длину любого сообщения, не заботясь о соблюдении границ массивов.

Инициализация безразмерных массивов поддерживается не только для одномерных массивов. В многомерном массиве размер самого левого измерения также можно не указывать. (Размеры по остальным измерениям обязательно должны быть указаны, так как это нужно компилятору для определения длины подмассивов, составляющих массив). Таким способом можно создавать таблицы переменного размера, компилятор автоматически выделит требуемую для них память. Например, объявление `sqr` как безразмерного массива выглядит так:

```
int sqr[][2] = {
    {1, 1},
    {2, 4},
    {3, 9},
    {4, 16},
    {5, 25},
    {6, 36},
    {7, 49},
    {8, 64},
    {9, 81},
    {10, 100}
};
```

Преимущество безразмерного объявления массива состоит в том, что можно изменять длину таблицы, не заботясь о размере массива.



Массивы переменной длины

Как уже упоминалось, в С89 размер массива должен быть объявлен с помощью константных выражений. Поэтому компилятор С89 устанавливает фиксированный размер массива, не изменяющийся в процессе выполнения программы. Однако это не относится к С99, в котором определено новое мощное средство: массивы переменной длины. Стандарт С99 позволяет в объявлении размера массива использовать любые выражения, в том числе такие, значение которых становится известным только во время выполнения. Объявленный таким образом массив называется *массивом переменной длины*. Однако переменную длину могут иметь только локальные массивы (т.е. видимые в блоке или в прототипе). Приведем пример массива переменной длины:

```
void f(int dim)
{
    char str[dim]; /* символьный массив переменной длины */

    /* ... */
}
```

Здесь размер массива `str` определяется значением переменной `dim`, которая передается в функцию `f()` как параметр. Таким образом, при каждом вызове `f()` создается массив `str` разной длины.

Массивы переменной длины добавлены в С99 главным образом для поддержки численных методов обработки данных. В программировании это средство распространено достаточно широко. Однако следует помнить, что стандарт С89 (и некоторые компиляторы С++) не поддерживает массивы переменной длины. Более подробно этот вопрос рассматривается в части II.



Приемы использования массивов и строк на примере игры в крестики-нолики

Представленный длинный пример иллюстрирует большое количество приемов использования строк. Рассматривается простая программа игры в крестики-нолики. Двухмерный массив используется в качестве матрицы, изображающей игральную доску.

Компьютер играет в очень простую игру. Когда наступает очередь хода компьютера, функция `get_computer_move()` просматривает матрицу в поиске незанятых ячеек. Если функция находит незанятую ячейку, она помещает туда символ `O`. Если незанятой ячейки нет, то функция выводит сообщение об окончании игры и прекращает работу программы. Функция `get_player_move()` спрашивает игрока, где он хочет поместить символ `X`. Верхний левый угол имеет координаты `(1, 1)`, а нижний правый — `(3, 3)`.

Массив `matrix`, содержащий матрицу игры, инициализирован символами пробела. Каждый ход, сделанный игроком или компьютером, заменяет символ пробела символом `X` или `O`. Это позволяет легко отобразить матрицу на экране.

После каждого хода вызывается функция `check()`, которая возвращает пробел, если победителя еще нет, или `X`, если победил игрок, или `O`, когда победил компьютер. Эта функция просматривает строки, столбцы и диагонали в поиске трех одинаковых символов (`X` или `O`) подряд.

Функция `disp_matrix()` отображает на экране текущее состояние игры. Обратите внимание на то, как существенно упрощает эту функцию инициализация матрицы пробелами.

Функции получают доступ к массиву `matrix` различными способами. Их стоит внимательно изучить для лучшего понимания приемов работы с массивами.

```
/* Простая игра в крестики-нолики. */
#include <stdio.h>
#include <stdlib.h>

char matrix[3][3]; /* матрица игры */
char check(void);
void init_matrix(void);
void get_player_move(void);
void get_computer_move(void);
void disp_matrix(void);

int main(void)
{
    char done;

    printf(" Это игра в крестики-нолики.\n");
    printf(" Вы будете играть против компьютера.\n");

    done = ' ';
    init_matrix();

    do {
        disp_matrix();
        get_player_move();
        done = check(); /* проверка, есть ли победитель */
        if(done != ' ') break; /* есть победитель */
        get_computer_move();
        done = check(); /* проверка, есть ли победитель */
    } while(done == ' ');

    if(done == 'X') printf("Вы победили!\n");
    else printf("Победил компьютер!!!\n");
    disp_matrix(); /* показ финальной позиции */

    return 0;
}

/* Инициализация матрицы игры */
void init_matrix(void)
{
    int i, j;

    for(i=0; i<3; i++)
        for(j=0; j<3; j++) matrix[i][j] = ' ';
}

/* Ход игрока. */
void get_player_move(void)
{
    int x, y;

    printf("Введите координаты X,Y вашего хода: ");
```

```

scanf("%d%c%d", &x, &y);
x--; y--;

if(matrix[x][y] != ' '){
printf("Неверный ход, попробуйте еще.\n");
get_player_move();
}
else matrix[x][y] = 'X';
}

/* Ход компьютера. */
void get_computer_move(void)
{
    int i, j;
    for(i=0; i<3; i++){
        for(j=0; j<3; j++){
            if(matrix[i][j]==' ') break;
            if(matrix[i][j]==' ') break;
        }
        /* Второй break нужен для выхода из цикла по i */
    }

    if(i*j == 9) {
        printf("Конец игры.\n");
        exit(0);
    }
    else
        matrix[i][j] = 'O';
}

/* Вывод матрицы на экран. */
void disp_matrix(void)
{
    int t;

    for(t=0; t<3; t++) {
        printf(" %c | %c | %c ", matrix[t][0],
            matrix[t][1], matrix[t][2]);
        if(t!=2) printf("\n---|---|---\n");
    }
    printf("\n");
}

/* Определение победителя. */
char check(void)
{
    int i;

    for(i=0; i<3; i++) /* проверка строк */
        if(matrix[i][0] == matrix[i][1] &&
            matrix[i][0] == matrix[i][2])
            return matrix[i][0];

    for(i=0; i<3; i++) /* проверка столбцов */
        if(matrix[0][i] == matrix[1][i] &&
            matrix[0][i] == matrix[2][i])
            return matrix[0][i];

    /* проверка диагоналей */

```

```

    if(matrix[0][0] == matrix[1][1] &&
        matrix[1][1] == matrix[2][2])
        return matrix[0][0];

    if(matrix[0][2] == matrix[1][1] &&
        matrix[1][1] == matrix[2][0])
        return matrix[0][2];

    return ' ';
}

```

Пояснение к программе. В функции `get_player_move()` с помощью библиотечной функции `scanf()` считываются с клавиатуры два целых числа `x` и `y`. Функция `scanf()` при считывании чисел предполагает, что во входном потоке они разделены пробелами (или пробельными символами), другие разделительные символы не допускаются. Однако многие пользователи привыкли к тому, что числа можно разделять, например, запятыми. (Собственно говоря, именно так и предлагается в подсказке, выдаваемой программой.) В приведенном примере символ, следующий непосредственно после первого числа, просто игнорируется, именно для этого в функции `scanf()` используется спецификатор формата `%*c`. Звездочка означает, что символ считывается из потока, но в память не записывается¹.

¹ Пояснение добавлено редактором. — *Прим. ред.*

Полный
справочник по



Глава 5

Указатели

Правильное понимание и использование указателей особенно необходимо для составления хороших программ на языке С. И вот почему. Во-первых, указатели являются средством, с помощью которого функция может изменять значения передаваемых в нее аргументов. Во-вторых, с помощью указателей выполняется динамическое распределение памяти. В-третьих, указатели позволяют повысить эффективность многих процедур. И наконец, они обеспечивают поддержку динамических структур данных, таких, например, как двоичные деревья и связанные списки.

Таким образом, указатели являются весьма мощным средством языка С. Но и весьма опасным. Например, если указатель содержит неправильное значение, программа может потерпеть крах. Указатели весьма опасны еще и потому, что легко ошибиться при их использовании. К тому же ошибки, связанные с неправильными значениями указателей, найти очень трудно.

■ Что такое указатели

Указатель — это переменная, значением которой является адрес некоторого объекта (обычно другой переменной) в памяти компьютера. Например, если одна переменная содержит адрес другой переменной, то говорят, что первая переменная *указывает* (ссылается) на вторую. Это иллюстрируется с помощью рис. 5.1.

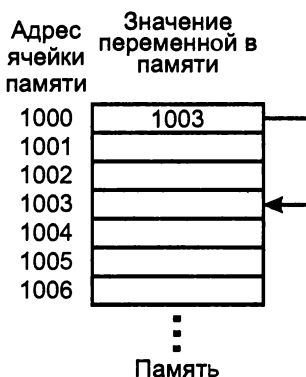


Рис. 5.1. Одна переменная ссылается на другую

■ Указательные переменные

Как известно, переменную, являющуюся указателем, нужно соответствующим образом объявить. Объявление указателя состоит из имени базового типа, символа * и имени переменной. Общая форма объявления указателя следующая:

тип **имя*;

Здесь *тип* — это базовый тип указателя, им может быть любой правильный тип. *Имя* определяет имя переменной-указателя.

Базовый тип указателя определяет тип объекта, на который указатель будет ссылаться. Фактически указатель любого типа может ссылаться на любое место в памяти. Однако выполняемые с указателем операции существенно зависят от его типа. Например, если объявлен указатель типа `int *`, компилятор предполагает, что любой

адрес, на который он ссылается, содержит переменную типа `int`, хотя это может быть и не так. Следовательно, объявляя указатель, необходимо убедиться, что его тип совместим с типом объекта, на который он будет ссылаться.

■ Операции для работы с указателями

Операции для работы с указателями рассматривались в главе 2. Приведем их обзор. В языке C определены две операции для работы с указателями: `*` и `&`. Оператор `&` — это унарный оператор, возвращающий адрес своего операнда. (Напомним, что унарный оператор имеет один операнд). Например, оператор

```
m = &count;
```

присваивает переменной `m` адрес переменной `count`. Можно сказать, что адрес — это номер первого байта участка памяти, в котором хранится переменная. Адрес и значение переменной — это совершенно разные понятия. Оператор `&` можно представить себе как оператор, возвращающий адрес объекта. Следовательно, предыдущий пример можно прочесть так: “переменной `m` присваивается адрес переменной `count`”.

Предположим, переменная `count` хранится в ячейке памяти под номером 2000, а ее значение равно 100. Тогда переменной `m` будет присвоено значение 2000.

Вторая операция для работы с указателями (ее знак, т.е. оператор, `*`) выполняет действие, обратное по отношению к `&`. Оператор `*` — это унарный оператор, возвращающий значение переменной, расположенной по указанному адресу. Например, если `m` содержит адрес переменной `count`, то оператор

```
q = *m;
```

присваивает переменной `q` значение переменной `count`. Таким образом, `q` получит значение 100, потому что по адресу 2000 расположена переменная `count`, которая имеет значение 100. Действие оператора `*` можно выразить словами “значение по адресу”, тогда предыдущий оператор может быть прочитан так: “`q` получает значение переменной, расположенной по адресу `m`”.

■ Указательные выражения

В общем случае выражения с указателями подчиняются тем же правилам, что и обычные выражения. В этом разделе рассматривается применение указательных выражений в операциях присваивания, преобразования типов, а также в операциях “указательной” арифметики.

Присваивание указателей

Указатель можно использовать в правой части оператора присваивания для присваивания его значения другому указателю. Если оба указателя имеют один и тот же тип, то выполняется простое присваивание, без преобразования типа. В следующем примере

```
#include <stdio.h>

int main(void)
{
    int x = 99;
    int *p1, *p2;

    p1 = &x;
```

```

p2 = p1;

/* печать значения x дважды */
printf("Значения по адресу p1 и p2 : %d %d\n", *p1, *p2);

/* печать адреса x дважды */
printf("Значения указателей p1 и p2: %p %p", p1, p2);

return 0;
}

```

после присваивания

```

p1 = &x;
p2 = p1;

```

оба указателя (p1 и p2) ссылаются на x. То есть, оба указателя ссылаются на один и тот же объект. Программа выводит на экран следующее:

```

Значения по адресу p1 и p2 : 99 99
Значения указателей p1 и p2: 0063FDF0 0063FDF0

```

Обратите внимание, для вывода значений указателей в функции printf() используется спецификатор формата %p, который выводит адреса в формате, используемом компилятором.

Допускается присваивание указателя одного типа указателю другого типа. Однако для этого необходимо выполнить явное преобразование типа указателя (операция приведения типов), которая рассматривается в следующем разделе.

Преобразование типа указателя

Указатель можно преобразовать к другому типу. Эти преобразования бывают двух видов: с использованием указателя типа void * и без его использования.

В языке C допускается присваивание указателя типа void * указателю любого другого типа (и наоборот) без явного преобразования типа указателя. Тип указателя void * используется, если тип объекта неизвестен. Например, использование типа void * в качестве параметра функции позволяет передавать в функцию указатель на объект любого типа, при этом сообщение об ошибке не генерируется. Также он полезен для ссылки на произвольный участок памяти, независимо от размещенных там объектов. Например, функция размещения malloc() (рассматривается далее в этой главе) возвращает значение типа void *, что позволяет использовать ее для размещения в памяти объектов любого типа.

В отличие от void *, преобразования всех остальных типов указателей должны быть всегда явными (т.е. должна быть указана операция приведения типов). Однако следует учитывать, что преобразование одного типа указателя к другому может вызвать непредсказуемое поведение программы. Например, в следующей программе делается попытка присвоить значение x переменной y посредством указателя p. При компиляции программы сообщение об ошибке не генерируется, однако результат работы программы неверен.

```

#include <stdio.h>

int main(void)
{
    double x = 100.1, y;
    int *p;

    /* В следующем операторе указателю на целое p присваивается

```

```

        значение, ссылающееся на double */
p = (int *) &x;

/* Следующий оператор работает не так, как ожидается */
y = *p;

/* Следующий оператор не выведет число 100.1 */
printf("значение x равно %f (Это не так!)", y);

return 0;
}

```

Обратите внимание на то, что операция приведения типов применяется в операторе присваивания адреса переменной `x` (он имеет тип `double *`) указателю `p`, тип которого `int *`. Преобразование типа выполнено корректно, однако программа работает не так, как ожидается (по крайней мере, в большинстве оболочек). Для разъяснения проблемы предположим, что переменная `int` занимает в памяти 4 байта, а `double` — 8 байтов. Указатель `p` объявлен как указатель на целую переменную (т.е. типа `int`), поэтому оператор присваивания

```

y = *p;

```

передаст переменной `y` только 4 байта информации, а не 8 байтов, необходимых для `double`. Несмотря на то, что `p` ссылается на объект `double`, оператор присваивания выполнит действие с объектом типа `int`, потому что `p` объявлен как указатель на `int`. Поэтому такое использование указателя `p` неправильное.

Приведенный пример подтверждает то, что операции с указателями выполняются в зависимости от базового типа указателей. Синтаксически допускается ссылка на объект с типом, отличным от типа указателя, однако при этом указатель будет “думать”, что он ссылается на объект своего типа. Таким образом, операции с указателями управляются типом указателя, а не типом объекта, на который он ссылается.

Разрешен еще один тип преобразований: преобразование целого в указатель и наоборот. В этом случае необходимо применить операцию приведения типов (явное преобразование типа). Однако пользоваться этим средством нужно очень осторожно, потому что при этом легко получить непредсказуемое поведение программы. Явное преобразование типа не обязательно, если преобразуется нуль, то есть нулевой указатель.

На заметку

*В языке C++ требуется явно указывать преобразование типа указателей, в том числе указателей типа `void *`. Поэтому многие программисты используют в языке C явное преобразование для совместимости с C++.*

Адресная арифметика

В языке C допустимы только две арифметические операции над указателями: суммирование и вычитание. Предположим, текущее значение указателя `p1` типа `int *` равно 2000. Предположим также, что переменная типа `int` занимает в памяти 2 байта. Тогда после операции увеличения

```

p1++;

```

указатель `p1` принимает значение 2002, а не 2001. То есть, при увеличении на 1 указатель `p1` будет ссылаться на следующее целое число. Это же справедливо и для операции уменьшения. Например, если `p1` равно 2000, то после выполнения оператора

```

p1--;

```

значение `p1` будет равно 1998.

Операции адресной арифметики подчиняются следующим правилам. После выполнения операции увеличения над указателем, данный указатель будет ссылаться на следующий объект своего базового типа. После выполнения операции уменьшения — на предыдущий объект. Применительно к указателям на `char`, операции адресной арифметики выполняются как обычные арифметические операции, потому что длина объекта `char` всегда равна 1. Для всех указателей адрес увеличивается или уменьшается на величину, равную размеру объекта того типа, на который они указывают. Поэтому указатель всегда ссылается на объект с типом, тождественным базовому типу указателя. Эта концепция иллюстрируется с помощью рис. 5.2.

Операции адресной арифметики не ограничены увеличением (инкрементом) и уменьшением (декрементом). Например, к указателям можно добавлять целые числа или вычитать из них целые числа. Выполнение оператора

```
■ p1 = p1 + 12;
```

“передвигает” указатель `p1` на 12 объектов в сторону увеличения адресов.

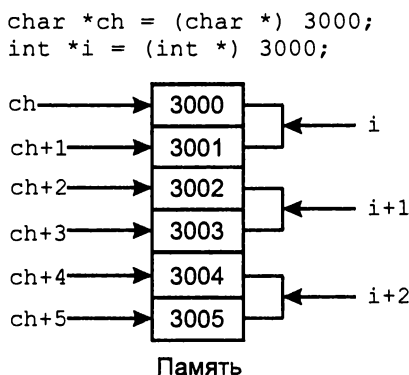


Рис. 5.2. Пример размещения в памяти переменных `char` (слева) и `int` (справа)

Кроме суммирования и вычитания указателя и целого, разрешена еще только одна операция адресной арифметики: можно вычитать два указателя. Благодаря этому можно определить количество объектов, расположенных между адресами, на которые указывают данные два указателя; правда, при этом считается, что тип объектов совпадает с базовым типом указателей. Все остальные арифметические операции запрещены. А именно: нельзя делить и умножать указатели, суммировать два указателя, выполнять над указателями побитовые операции, суммировать указатель со значениями, имеющими тип `float` или `double` и т.д.

Сравнение указателей

Стандартом C допускается сравнение двух указателей. Например, если объявлены два указателя `p` и `q`, то следующий оператор является правильным:

```
■ if (p < q) printf("p ссылается на меньший адрес, чем q");
```

Как правило, сравнение указателей может оказаться полезным, только тогда, когда два указателя ссылаются на общий объект, например, на массив. В качестве примера рассмотрим программу с двумя стековыми функциями, предназначенными для записи и считывания целых чисел. Стек — это список, использующий систему доступа “первым вошел — последним вышел”. Иногда стек сравнивают со стопкой тарелок на

столе: первая, поставленная на стол, будет взята последней. Стеки часто используются в компиляторах, интерпретаторах, программах обработки крупноформатных таблиц и в других системных программах. Для создания стека необходимы две функции: `push()` и `pop()`. Функция `push()` заносит числа в стек, а `pop()` — извлекает их. В данном примере эти функции используются в `main()`. При вводе числа с клавиатуры, программа помещает его в стек. Если ввести 0, то число извлекается из стека. Программа завершает работу при вводе -1.

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

#define SIZE 50

void push(int i);
int pop(void);

int *tos, *p1, stack[SIZE];

int main(void)
{
    int value;

    tos = stack; /* tos ссылается на основание стека */
    p1 = stack; /* инициализация p1 */

    do {
        printf("Введите значение: ");
        scanf("%d", &value);

        if(value != 0) push(value);
        else printf("значение на вершине равно %d\n", pop());
    } while(value != -1);

    return 0;
}

void push(int i)
{
    p1++;
    if(p1 == (tos+SIZE)) {
        printf("Переполнение стека.\n");
        exit(1);
    }
    *p1 = i;
}

int pop(void)
{
    if(p1 == tos) {
        printf("Стек пуст.\n");
        exit(1);
    }
    p1--;
    return *(p1+1);
}
```

Стек хранится в массиве `stack`. Сначала указатели `p1` и `tos` устанавливаются на первый элемент массива `stack`. В дальнейшем `p1` ссылается на верхний элемент сте-

ка, а `tos` продолжает хранить адрес основания стека. После инициализации стека используются функции `push()` и `pop()`. Они выполняют запись в стек и считывание из него, проверяя каждый раз соблюдение границы стека. В функции `push()` проверяется, что указатель `p1` не превышает верхней границы стека `tos+SIZE`. Это предотвращает переполнение стека. В функции `pop()` проверяется, что указатель `p1` не выходит за нижнюю границу стека.

В операторе `return` функции `pop()` скобки необходимы потому, что без них оператор

```
return *p1+1;
```

вернул бы значение, расположенное по адресу `p1`, увеличенное на 1, а не значение по адресу `p1+1`.



Указатели и массивы

Понятия указателей и массивов тесно связаны. Рассмотрим следующий фрагмент программы:

```
char str[80], *p1;  
p1 = str;
```

Здесь `p1` указывает на первый элемент массива `str`. Обратиться к пятому элементу массива `str` можно с помощью любого из двух выражений:

```
str[4]  
*(p1+4)
```

Массив начинается с нуля. Поэтому для пятого элемента массива `str` нужно использовать индекс 4. Можно также увеличить `p1` на 4, тогда он будет указывать на пятый элемент. (Напомним, что имя массива без индекса возвращает адрес первого элемента массива.)

В языке C существуют два метода обращения к элементу массива: адресная арифметика и индексация массива. Стандартная запись массивов с индексами наглядна и удобна в использовании, однако с помощью адресной арифметики иногда удастся сократить время доступа к элементам массива. Поэтому адресная арифметика часто используется в программах, где существенную роль играет быстродействие.

В следующем фрагменте программы приведены две версии функции `putstr()`, выводящей строку на экран. В первой версии используется индексация массива, а во второй — адресная арифметика:

```
/* Индексация указателя s как массива */  
void putstr(char *s)  
{  
    register int t;  
    for(t=0; s[t]; ++t) putchar(s[t]);  
}  
  
/* Использование адресной арифметики */  
void putstr(char *s)  
{  
    while(*s) putchar(*s++);  
}
```

Большинство профессиональных программистов сочтут вторую версию более наглядной и удобной. Для большинства компиляторов она также более быстродействующая. Поэтому в процедурах такого типа приемы адресной арифметики используются довольно часто.

Массивы указателей

Как и объекты любых других типов, указатели могут быть собраны в массив. В следующем операторе объявлен массив из 10 указателей на объекты типа `int`:

```
int *x[10];
```

Для присвоения, например, адреса переменной `var` третьему элементу массива указателей, необходимо написать:

```
x[2] = &var;
```

В результате этой операции, следующее выражение принимает то же значение, что и `var`:

```
*x[2]
```

Для передачи массива указателей в функцию используется тот же метод, что и для любого другого массива: имя массива без индекса записывается как формальный параметр функции. Например, следующая функция может принять массив `x` в качестве аргумента:

```
void display_array(int *q[])
{
    int t;

    for(t=0; t<10; t++)
        printf("%d ", *q[t]);
}
```

Необходимо помнить, что `q` — это не указатель на целые, а указатель на массив указателей на целые. Поэтому параметр `q` нужно объявить как массив указателей на целые. Нельзя объявить `q` просто как указатель на целые, потому что он представляет собой указатель на указатель.

Массивы указателей часто используются при работе со строками. Например, можно написать функцию, выводящую нужную строку с сообщением об ошибке по индексу `num`:

```
void syntax_error(int num)
{
    static char *err[] = {
        "Нельзя открыть файл\n",
        "Ошибка при чтении\n",
        "Ошибка при записи\n",
        "Некачественный носитель\n"
    };

    printf("%s", err[num]);
}
```

Массив `err` содержит указатели на строки с сообщениями об ошибках. Здесь строковые константы в выражении инициализации создают указатели на строки. Аргументом функции `printf()` служит один из указателей массива `err`, который в соответствии с индексом `num` указывает на нужную строку с сообщением об ошибке. Например, если в функцию `syntax_error()` передается `num` со значением 2, то выводится сообщение Ошибка при записи.

Отметим, что аргумент командной строки `argv` (см. главу 6) также является массивом указателей на строковые константы.

Многоуровневая адресация

Иногда указатель может ссылаться на указатель, который ссылается на число. Это называется *многоуровневой адресацией*. Иногда применение таких указателей существенно усложняет программу, делает ее плохо читаемой и подверженной ошибкам. Рис. 5.3 иллюстрирует концепцию многоуровневой адресации. На рисунке видно, что значением “нормального” указателя является адрес объекта, содержащего нужное значение. В случае двухуровневой адресации первый указатель содержит адрес второго указателя, который содержит адрес объекта с нужным значением.

Многоуровневая адресация может иметь сколько угодно уровней, однако уровни глубже второго, т.е. указатели более глубокие, чем “указатели на указатели” применяются крайне редко. Дело в том, что при использовании таких указателей часто встречаются концептуальные ошибки из-за того, что смысл таких указателей представить трудно.

На заметку

Не следует путать многоуровневую адресацию с многоуровневыми структурами данных, использующими указатели, такими, например, как связанные списки. Это фундаментально различные концепции.

Переменная, являющаяся указателем на указатель, должна быть соответствующим образом объявлена. Это делается с помощью двух звездочек перед именем переменной. Например, в следующем операторе `newbalance` объявлена как указатель на указатель на переменную типа `float`:

```
float **newbalance;
```

Следует хорошо понимать, что `newbalance` — это не указатель на число типа `float`, а указатель на указатель на число типа `float`.

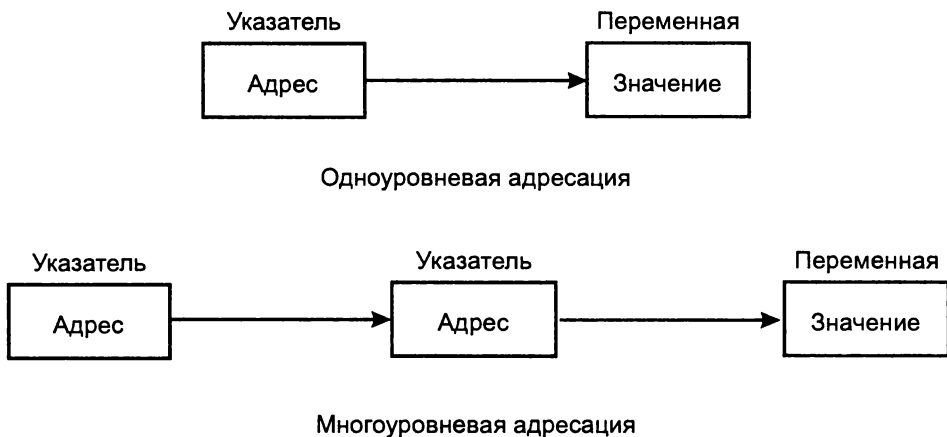


Рис. 5.3. Одноуровневая и многоуровневая адресации

При двухуровневой адресации для доступа к значению объекта нужно поставить перед идентификатором две звездочки:

```
#include <stdio.h>

int main(void)
{
    int x, *p, **q;

    x = 20;
```

```

    p = &x;
    q = &p;

    printf("%d", **q); /* печать значения x */

    return 0;
}

```

Здесь `p` объявлена как указатель на целое, а `q` — как указатель на указатель на целое. Функция `printf()` выводит на экран число 10.

■ Инициализация указателей

После объявления нестатического локального указателя до первого присвоения он содержит неопределенное значение. (Глобальные и статические локальные указатели при объявлении неявно инициализируются нулем.) Если попытаться использовать указатель перед присвоением ему нужного значения, то скорее всего он мгновенно разрушит программу или всю операционную систему. Это очень досадная ошибка.

При работе с указателями большинство программистов придерживаются следующего важного соглашения: указатель, не ссылающийся в текущий момент времени должным образом на конкретный объект, должен содержать нулевое значение. Нуль используется потому, что С гарантирует отсутствие чего-либо по нулевому адресу. Следовательно, если указатель равен нулю, то это значит, во-первых, что он ни на что не ссылается, а во-вторых — что его сейчас нельзя использовать.

Указателю можно задать нулевое значение, присвоив ему 0. Например, следующий оператор инициализирует `p` нулем:

```

char *p = 0;

```

Дополнительно к этому во многих заголовочных файлах языка С, например, в `<stdio.h>` определен макрос `NULL`, являющийся нулевой указательной константой. Поэтому в программах на С часто можно увидеть следующее присваивание:

```

p = NULL;

```

Однако равенство указателя нулю не делает его абсолютно “безопасным”. Использование нуля в качестве признака неподготовленности указателя — это только соглашение программистов, но не правило языка С. В следующем примере компиляция пройдет без ошибки, а результат, тем не менее, будет неправильным:

```

int *p = 0;
*p = 10; /* ошибка! */

```

В этом случае присваивание посредством `p` будет присваиванием по нулевому адресу, что обычно вызывает разрушение программы.

Во многих процедурах для повышения эффективности программы можно использовать то, что нулевой указатель заведомо считается неподготовленным для использования. Например, можно использовать нулевой указатель как признак конца массива указателей (по аналогии с нулевым терминатором строки). Процедура, использующая массив указателей, таким образом узнает о конце массива. Такой подход иллюстрируется в таком примере. Просматривая список имен, функция `search()` определяет, есть ли в этом списке заданное имя.

```

#include <stdio.h>
#include <string.h>

int search(char *p[], char *name);

```

```

char *names[] = {
    "Сергей",
    "Юрий",
    "Ольга",
    "Игорь",
    NULL}; /* Нулевая константа кончает список */

int main(void){
    if(search(names, "Ольга") != -1)
        printf("Ольга есть в списке.\n");

    if(search(names, "Павел") == -1)
        printf("Павел в списке не найден.\n");

    return 0;
}

/* Просмотр имен */
int search(char *p[], char *name)
{
    register int t;

    for(t=0; p[t]; ++t)
        if(!strcmp(p[t], name)) return t;

    return -1; /* имя не найдено */
}

```

В функцию `search()` передаются два параметра. Первый из них, `p` — массив указателей на строки, представляющие собой имена из списка. Второй параметр `name` является указателем на строку с заданным именем. Функция `search()` просматривает массив указателей, пока не найдет строку, совпадающую со строкой, на которую указывает `name`. Итерации цикла `for` повторяются до тех пор, пока не произойдет совпадение имен, или не встретится нулевой указатель. Конец массива отмечен нулевым указателем, поэтому при достижении конца массива управляющее условие цикла примет значение ЛОЖЬ. Иными словами, `p[t]` имеет значение ЛОЖЬ, когда `p[t]` является нулевым указателем. В рассмотренном примере именно это и происходит, когда идет поиск имени “Павел”, которого в списке нет.

В программах на С указатель типа `char *` часто инициализируют строковой константой (как в предыдущем примере). Рассмотрим следующий пример:

```

char *p = "тестовая строка";

```

Переменная `p` является указателем, а не массивом. Поэтому возникает логичный вопрос: где хранится строковая константа “тестовая строка”? Так как `p` не является массивом, она не может храниться в `p`, тем не менее, она где-то записана. Чтобы ответить на этот вопрос, нужно знать, что происходит, когда компилятор встречает строковую константу. Компилятор создает так называемую *таблицу строк*, в ней он сохраняет строковые константы, которые встречаются ему по ходу чтения текста программы. Следовательно, когда встречается объявление с инициализацией, компилятор сохраняет строку “тестовая строка” в таблице строк, а в указатель `p` записывает ее адрес. Дальше в программе указатель `p` может быть использован как любая другая строка. Это иллюстрируется следующим примером:

```

#include <stdio.h>
#include <string.h>

char *p = "тестовая строка";

```

```

int main(void)
{
    register int t;

    /* печать строки слева направо и справа налево */
    printf(p);
    for(t=strlen(p)-1; t>-1; t--) printf("%c", p[t]);

    return 0;
}

```



Указатели на функции

*Указатели на функции*¹ — очень мощное средство языка С. Хотя нельзя не отметить, что это весьма трудный для понимания термин. Функция располагается в памяти по определенному адресу, который можно присвоить указателю в качестве его значения. Адресом функции является ее точка входа. Именно этот адрес используется при вызове функции. Так как указатель хранит адрес функции, то она может быть вызвана с помощью этого указателя. Он позволяет также передавать ее другим функциям в качестве аргумента.

В программе на С адресом функции служит ее имя без скобок и аргументов (это похоже на адрес массива, который равен имени массива без индексов). Рассмотрим следующую программу, в которой сравниваются две строки, введенные пользователем. Обратите внимание на объявление функции `check()` и указатель `p` внутри `main()`. Указатель `p`, как вы увидите, является указателем на функцию.

```

#include <stdio.h>
#include <string.h>

void check(char *a, char *b,
           int (*cmp)(const char *, const char *));

int main(void)
{
    char s1[80], s2[80];
    int (*p)(const char *, const char *);
        /* указатель на функцию */

    p = strcmp;
        /* присваивает адрес функции strcmp указателю p */

    printf("Введите две строки.\n");
    gets(s1);
    gets(s2);

    check(s1, s2, p); /* передает адрес функции strcmp
                        посредством указателя p */

    return 0;
}

```

¹ Иногда их называют просто *указателями функций*. Но следует помнить, что в языках программирования под этим термином подразумевается также средство обращения к подпрограмме-функции или встроенной функции, имеющее конструкцию *<имя-функции>* (*<список-аргументов>*). — Прим. ред.

```

void check(char *a, char *b,
           int (*cmp)(const char *, const char *))
{
    printf("Проверка на совпадение.\n");
    if(!(*cmp)(a, b)) printf("Равны");
    else printf("Не равны");
}

```

Проанализируем эту программу подробно. В первую очередь рассмотрим объявление указателя `p` в `main()`:

```
int (*p)(const char *,const char *);
```

Это объявление сообщает компилятору, что `p` — это указатель на функцию, имеющую два параметра типа `const char *` и возвращающую значение типа `int`. Скобки вокруг `p` необходимы для правильной интерпретации объявления компилятором. Подобная форма объявления используется также для указателей на любые другие функции, нужно лишь внести изменения в зависимости от возвращаемого типа и параметров функции.

Теперь рассмотрим функцию `check()`. В ней объявлены три параметра: два указателя на символьный тип (`a` и `b`) и указатель на функцию `cmp`. Обратите внимание на то, что указатель функции `cmp` объявлен в том же формате, что и `p`. Поэтому в `cmp` можно хранить значение указателя на функцию, имеющую два параметра типа `const char *` и возвращающую значение `int`. Как и в объявлении `p`, круглые скобки вокруг `*cmp` необходимы для правильной интерпретации этого объявления компилятором.

Вначале в программе указателю `p` присваивается адрес стандартной библиотечной функции `strcmp()`, которая сравнивает строки. Потом программа просит пользователя ввести две строки и передает указатели на них функции `check()`, которая их сравнивает. Внутри `check()` выражение

```
(*cmp)(a, b)
```

вызывает функцию `strcmp()`, на которую указывает `cmp`, с аргументами `a` и `b`. Скобки вокруг `*cmp` обязательны. Существует и другой, более простой, способ вызова функции с помощью указателя:

```
cmp(a, b);
```

Однако первый способ используется чаще (и мы рекомендуем использовать именно его), потому что при втором способе вызова указатель `cmp` очень похож на имя функции, что может сбить с толку читающего программу. В то же время у первого способа записи есть свои преимущества, например, хорошо видно, что функция вызывается с помощью указателя на функцию, а не имени функции. Следует отметить, что первоначально в С был определен именно первый способ вызова.

Вызов функции `check()` можно записать, используя непосредственно имя `strcmp()`:

```
check(s1, s2, strcmp);
```

В этом случае вводить в программу дополнительный указатель `p` нет необходимости.

У читателя может возникнуть вопрос: какая польза от вызова функции с помощью указателя на функцию? Ведь в данном случае никаких преимуществ не достигнуто, этим мы только усложнили программу. Тем не менее, во многих случаях оказывается более выгодным передать имя функции как параметр или даже создать массив функций. Например, в программе интерпретатора синтаксический анализатор (программа, анализирующая выражения) часто вызывает различные вспомогательные функции, такие как вычисление математических функций, процедуры ввода-вывода и т.п. В таких случаях чаще всего создают список функций и вызывают их с помощью индексов.

Альтернативный подход — использование оператора switch с длинным списком меток case — делает программу более громоздкой и подверженной ошибкам.

В следующем примере рассматривается расширенная версия предыдущей программы. В этой версии функция check() устроена так, что может выполнять разные операции над строками s1 и s2 (например, сравнивать каждый символ с соответствующим символом другой строки или сравнивать числа, записанные в строках) в зависимости от того, какая функция указана в списке аргументов. Например, строки "0123" и "123" отличаются, однако представляют одно и то же числовое значение.

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

void check(char *a, char *b,
           int (*cmp)(const char *, const char *));
int compvalues(const char *a, const char *b);

int main(void)
{
    char s1[80], s2[80];

    printf("Введите два значения или две строки.\n");
    gets(s1);
    gets(s2);

    if(isdigit(*s1)) {
        printf("Проверка значений на равенство.\n");
        check(s1, s2, compvalues);
    }
    else {
        printf("Проверка строк на равенство.\n");
        check(s1, s2, strcmp);
    }

    return 0;
}

void check(char *a, char *b,
           int (*cmp)(const char *, const char *))
{
    if(!(*cmp)(a, b)) printf("Равны");
    else printf("Не равны");
}

int compvalues(const char *a, const char *b)
{
    if(atoi(a) == atoi(b)) return 0;
    else return 1;
}
```

Если в этом примере ввести первый символ первой строки как цифру, то check() использует compvalues(), в противном случае — strcmp(). Функция check() вызывает ту функцию, имя которой указано в списке аргументов при вызове check(), поэтому она в разных ситуациях может вызывать разные функции. Ниже приведены результаты работы этой программы в двух случаях:

```
Введите два значения или две строки.  
тест  
тест  
Проверка строк на равенство.  
Равны
```

```
Введите два значения или две строки.  
0123  
123  
Проверка значений на равенство.  
Равны
```

Сравнение строк 0123¹ и 123 показывает равенство их значений.



Функции динамического распределения

Указатели используются для динамического выделения памяти компьютера для хранения данных. *Динамическое распределение* означает, что программа выделяет память для данных во время своего выполнения. Память для глобальных переменных выделяется во время компиляции, а для нестатических локальных переменных — в стеке. Во время выполнения программы ни глобальным, ни локальным переменным не может быть выделена дополнительная память. Но довольно часто такая необходимость возникает, причем объем требуемой памяти заранее неизвестен. Такое случается, например, при использовании динамических структур данных, таких как связанные списки или двоичные деревья. Такие структуры данных при выполнении программы расширяются или сокращаются по мере необходимости. Для реализации таких структур в программе нужны средства, способные по мере необходимости выделять и освобождать для них память.

Память, выделяемая в С функциями динамического распределения данных, находится в т.н. *динамически распределяемой области памяти (heap)*². Динамически распределяемая область памяти — это свободная область памяти, не используемая программой, операционной системой или другими программами. Размер динамически распределяемой области памяти заранее неизвестен, но как правило в ней достаточно памяти для размещения данных программы. Большинство компиляторов поддерживают библиотечные функции, позволяющие получить текущий размер динамически распределяемой области памяти, однако эти функции не определены в Стандарте С. Хотя размер динамически распределяемой области памяти очень большой, все же она конечна и может быть исчерпана.

Основу системы динамического распределения в С составляют функции `malloc()` и `free()`. Эти функции работают совместно. Функция `malloc()` выделяет память, а `free()` — освобождает ее. Это значит, что при каждом запросе функция `malloc()` выделяет требуемый участок свободной памяти, а `free()` освобождает его, то есть возвращает системе. В программу, использующую эти функции, должен быть включен заголовочный файл `<stdlib.h>`.

Прототип функции `malloc()` следующий:

```
void *malloc(size_t количество_байтов);
```

¹ Обратите внимание, что в языке С нулем начинаются восьмеричные константы. Если бы эта запись была в выражении, то 0123 не было бы равно 123. Однако здесь функция `atoi()` интерпретирует это число как десятичное. — *Прим. перев.*

² Применяются и другие названия: *динамическая область*, *динамически распределяемая область*, *куча*, *неупорядоченный массив (данных)*. — *Прим. ред.*

Здесь *количество_байтов* — размер памяти, необходимой для размещения данных. (Тип `size_t` определен в `<stdlib.h>` как некоторый целый без знака.) Функция `malloc()` возвращает указатель типа `void *`, поэтому его можно присвоить указателю любого типа. При успешном выполнении `malloc()` возвращает указатель на первый байт непрерывного участка памяти, выделенного в динамически распределяемой области памяти. Если в динамически распределяемой области памяти недостаточно свободной памяти для выполнения запроса, то память не выделяется и `malloc()` возвращает нуль.

При выполнении следующего фрагмента программы выделяется непрерывный участок памяти объемом 1000 байтов:

```
char *p;  
p = malloc(1000); /* выделение 1000 байтов */
```

После присвоения указатель `p` ссылается на первый из 1000 байтов выделенного участка памяти.

В следующем примере выделяется память для 50 целых. Для повышения мобильности (переносимости) программы с одной машины на другую) используется оператор `sizeof`.

```
int *p;  
p = malloc(50 * sizeof (int) );
```

Поскольку динамически распределяемая область памяти не бесконечна, при каждом размещении данных необходимо проверять, состоялось ли оно. Если `malloc()` не смогла по какой-либо причине выделить требуемый участок памяти, то она возвращает нуль. В следующем примере показано, как выполняется проверка успешности размещения:

```
p = malloc(100);  
if(!p) {  
    printf("Нехватка памяти.\n");  
    exit(1);  
}
```

Конечно, вместо выхода из программы `exit()` можно поставить какой-либо обработчик ошибки. Обязательным здесь можно назвать лишь требование не использовать указатель `p`, если он равен нулю.

Функция `free()` противоположна функции `malloc()` в том смысле, что она возвращает системе участок памяти, выделенный ранее с помощью функции `malloc()`. Иными словами, она освобождает участок памяти, который может быть вновь использован функцией `malloc()`. Функция `free()` имеет следующий прототип:

```
void free(void *p);
```

Здесь `p` — указатель на участок памяти, выделенный перед этим функцией `malloc()`. Функцию `free()` ни в коем случае нельзя вызывать с неправильным аргументом, это мгновенно разрушит всю систему распределения памяти.

Подсистема динамического распределения в С используется совместно с указателями для создания различных программных конструкций, таких как связные списки и двоичные деревья. Несколько примеров использования таких конструкций приведены в части IV. Здесь рассматривается другое важное применение динамического размещения: размещение массивов.

Динамическое выделение памяти для массивов

Довольно часто возникает необходимость выделить память динамически, используя `malloc()`, но работать с этой памятью удобнее так, будто это массив, который можно индексировать. В этом случае нужно создать *динамический массив*. Сделать это несложно, потому что каждый указатель можно индексировать как массив. В следующем примере одномерный динамический массив содержит строку:


```

/* Динамическое размещение строки, строка вводится
   пользователем, а затем распечатывается справа налево. */
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int main(void)
{
    char *s;
    register int t;

    s = malloc(80);

    if(!s) {
        printf("Требуемая память не выделена.\n");
        exit(1);
    }

    gets(s);
    for(t=strlen(s)-1; t>=0; t--) putchar(s[t]);
    free(s);

    return 0;
}

```

Перед первым использованием *s* программа проверяет, успешно ли прошло выделение памяти. Эта проверка необходима для предотвращения случайного использования нулевого указателя. Обратите внимание на то, что указатель *s* используется в функции *gets()*, а также при выводе на экран (но на этот раз уже как обыкновенный массив).

Можно также динамически выделить память для многомерного массива. Для этого нужно объявить указатель, определяющий все, кроме самого левого измерения массива. В следующем примере¹ двумерный динамический массив содержит таблицу чисел от 1 до 10 в степенях 1, 2, 3 и 4.

```

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

int pwr(int a, int b);

int main(void)
{
    /* Объявление указателя на массив из 10 строк
       в которых хранятся целые числа (int). */
    int (*p)[10];

    register int i, j;

    /* выделение памяти для массива 4x10 */
    p = malloc(40*sizeof(int));

    if(!p) {
        printf("Требуемая память не выделена.\n");
        exit(1);
    }
}

```

¹ В примере динамически размещается только левое измерение массива. Однако это нетрудно сделать и для всех измерений, объявив указатель ***p* и разместив каждое измерение отдельно. Такой прием особенно удобен при написании функции, один из аргументов которой — двумерный массив с неизвестными заранее размерами измерений. — *Прим. перев.*

```

    for(j=1; j<11; j++)
        for(i=1; i<5; i++) p[i-1][j-1] = pwr(j, i);

    for(j=1; j<11; j++) {
        for(i=1; i<5; i++) printf("%10d ", p[i-1][j-1]);
        printf("\n");
    }

    return 0;
}

/* Возведение чисел в степень. */
pwr(int a, int b)
{
    register int t=1;

    for(; b; b--) t = t*a;
    return t;
}

```

Программа выводит на экран следующее:

1	1	1	1
2	4	8	16
3	9	27	81
4	16	64	256
5	25	125	625
6	36	216	1296
7	49	343	2401
8	64	512	4096
9	81	729	6561
10	100	1000	10000

Указатель `p` в главной программе (`main()`) объявлен как

```
int (*p)[10];
```

Следует отметить, что скобки вокруг `*p` обязательны. Такое объявление означает, что `p` указывает на массив из 10 целых. Если увеличить указатель `p` на 1, то он будет указывать на следующие 10 целых чисел. Таким образом, `p` — это указатель на двумерный массив с 10 числами в каждой строке. Поэтому `p` можно индексировать как обычный двумерный массив. Разница только в том, что здесь память выделена с помощью `malloc()`, а для обыкновенного массива память выделяет компилятор.

Как упоминалось ранее, в C++ нужно преобразовывать типы указателей явно. Поэтому чтобы данная программа была правильной и в C, и в C++, необходимо выполнить явное приведение типа значения, возвращаемого функцией `malloc()`. Для этого строчку, в которой указателю `p` присваивается это значение, нужно переписать следующим образом:

```
p = (int (*)[10]) malloc(40*sizeof(int));
```

Многие программисты используют явное преобразование типов указателей для обеспечения совместимости с C++.



Указатели с квалификатором `restrict`

Стандарт C99 дополнительно вводит новый квалификатор типа `restrict`, применимый только для указателей. Подробно этот спецификатор обсуждается в части II, здесь приведено только его краткое описание.

Если указатель объявлен с квалификатором `restrict`, то к объекту, на который он ссылается, можно обратиться только с помощью этого указателя. Обращение к объекту с помощью другого указателя возможно только в том случае, если другой указатель основан на первом. Таким образом, доступ к объекту можно получить только с помощью выражений, основанных на указателе с квалификатором `restrict`. Указатели `restrict` используются главным образом как параметры функции или совместно с `malloc()`. Если указатель объявлен с квалификатором `restrict`, компилятор способен лучше оптимизировать некоторые процедуры. Например, если два параметра функции определены как указатели с квалификатором `restrict`, то это сообщает компилятору о том, что они указывают на два разных (не пересекающихся) объекта. Квалификатор `restrict` не изменяет семантику программы.



Трудности при работе с указателями

Ничто не может доставить больше неприятностей, чем “дикий” указатель! Указатели похожи на обоюдоострое оружие: их возможности огромны, однако обезвредить ошибки в них особенно трудно.

Ошибочный указатель трудно найти потому, что ошибка в самом указателе никак себя не проявляет. Проблемы возникают при попытке обратиться к объекту с помощью этого указателя. Если значение указателя неправильное, то программа с его помощью обращается к произвольной ячейке памяти. При чтении в программу попадают неправильные данные, а при записи искажаются другие данные, хранящиеся в памяти, или портится участок программы, не имеющий никакого отношения к ошибочному указателю. В обоих случаях ошибка может не проявиться вовсе или проявиться позже в форме, никак не указывающей на ее причину.

Поскольку ошибки, связанные с указателями, особенно трудно обезвредить, при работе с указателями следует соблюдать особую осторожность. Рассмотрим некоторые ошибки, наиболее часто возникающие при работе с указателями. Классический пример — *неинициализированный указатель*:

```
/* Эта программа содержит ошибку. */
int main(void)
{
    int x, *p;

    x = 10;
    *p = x; /ошибка, p не инициализирован */

    return 0;
}
```

Эта программа присваивает значение 10 некоторой неизвестной области памяти. Рассмотрим, почему это происходит. Хотя указателю `p` не было присвоено никакого значения, но в момент выполнения операции `*p = x` он имел некоторое (совершенно произвольное!) значение. Поэтому здесь имела место попытка выполнить операцию записи в область памяти, на которую указывал данный указатель. В небольших программах такая ошибка часто остается незамеченной, потому что если программа и данные занимают немного места, то “выстрел наугад” скорее всего будет “промахом”. С увеличением размера программы вероятность “попасть” в нее возрастает.

В таком простом случае большинство компиляторов выводят предупреждение о том, что используется неинициализированный указатель. Однако подобная ошибка может произойти и в более завуалированном виде, тогда компилятор не сможет распознать ее.

Вторая распространенная ошибка заключается в простом недоразумении при использовании указателя:

```

/* Эта программа содержит ошибку */
#include <stdio.h>

int main(void)
{
    int x, *p;

    x = 10;
    p = x;

    printf("%d", *p);

    return 0;
}

```

Вызов `printf()` не выводит на экран значение `x`, равное 10. Выводится произвольная величина, потому что оператор

```
p = x;
```

записан неправильно. Он присваивает значение 10 указателю, однако указатель должен содержать адрес, а не значение. Правильный оператор выглядит так:

```
p = &x;
```

Большинство компиляторов при попытке присвоить указателю `p` значение `x` выведут предупреждающее сообщение, но, как и в предыдущем примере, компилятор не сможет распознать эту ошибку в более завуалированном виде.

Еще одна типичная ошибка происходит иногда при неправильном понимании принципов расположения переменных в памяти. Программисту ничего не известно о том, как используемые им данные располагаются в памяти, будут ли они расположены так же при следующем выполнении программы или как их расположат другие компиляторы. Поэтому сравнивать одни указатели с другими недопустимо. Например, программа

```

char s[80], y[80];
char *p1, *p2;

p1 = s;
p2 = y;
if(p1 < p2)...

```

в общем случае неправильна. (В некоторых необычных ситуациях иногда определяют относительное положение переменных, но это делают очень редко.)

Похожая ошибка возникает, когда делается необоснованное предположение о расположении массивов. Иногда, предполагая, что массивы расположены рядом, пытаются обращаться к ним с помощью одного и того же указателя, например:

```

int first[10], second[10];
int *p, t;

p = first;
for(t=0; t<20; t++) *p++ = t;

```

Так присваивать значения массивам `first` и `second` нельзя. Если компилятор разместит массивы рядом, это может и не привести к неправильному результату. Однако подобная ошибка особенно неприятна тем, что при проверке она может остаться незамеченной, а потом компилятор будет размещать массивы по-другому и программа выполнится неправильно.

В следующей программе приведен пример очень опасной ошибки. Постарайтесь сами найти ее, не подсматривая в последующее объяснение.

```

/* Это программа с ошибкой */
#include <string.h>

```

```
#include <stdio.h>

int main(void)
{
    char *p1;
    char s[80];

    p1 = s;
    do {
        gets(s); /* чтение строки */

        /* печать десятичного эквивалента
           каждого символа */
        while(*p1) printf(" %d", *p1++);

    }while(strcmp(s, "выполнено"));

    return 0;
}
```

Программа печатает значения символов ASCII, находящихся в строке *s*. Печать осуществляется с помощью *p1*, указывающего на *s*. Ошибка состоит в том, что указателю *p1* присвоено значение *s* только один раз, перед циклом. В первой итерации *p1* правильно проходит по символам строки *s*, однако в следующей итерации он начинает не с первого символа, а с того, которым закончил в предыдущей итерации. Так что во второй итерации *p1* может указывать на середину второй строки, если она длиннее первой, или же вообще на конец остатка первой строки. Исправленная версия программы записывается так:

```
/* Это правильная программа */
#include <string.h>
#include <stdio.h>

int main(void)
{
    char *p1;
    char s[80];

    do {
        p1 = s; /* установка p1 в начало строки s */
        gets(s); /* чтение строки */

        /* печать десятичного эквивалента
           каждого символа */
        while(*p1) printf(" %d", *p1++);

    }while(strcmp(s, "выполнено"));

    return 0;
}
```

При такой записи указатель *p1* в начале каждой итерации устанавливается на первый символ строки *s*. Об этом необходимо всегда помнить при повторном использовании указателей.

То, что неправильные указатели могут быть очень “коварными”, не может служить причиной отказа от их использования. Следует лишь быть осторожным и внимательно проанализировать каждое применение указателя в программе.

Полный
справочник по



Глава 6

Функции

Функции — это строительные элементы языка С и то место, в котором выполняется вся работа программы. В этой главе изучаются свойства функций, в том числе их аргументы, возвращаемые значения, прототипы, а также рекурсия.

Общий вид функции

В общем виде функция выглядит следующим образом:

```
возвр-тип имя-функции(список параметров)
{
    тело функции
}
```

возвр-тип определяет тип данного, возвращаемого функцией¹. Функция может возвращать любой тип данных, за исключением массивов *список параметров* — это список, элементы которого отделяются друг от друга запятыми. Каждый такой элемент состоит из имени переменной и ее типа данных. При вызове функции параметры принимают значения аргументов. Функция может быть и без параметров, тогда их список будет пустым. Такой пустой список можно указать в явном виде, поместив для этого внутри скобок ключевое слово `void`.

В объявлениях (декларациях) переменных можно объявить (декларировать) несколько переменных одного и того же типа, используя для этого список одних только имен, элементы которого отделены друг от друга запятыми. А все параметры функций, наоборот, должны объявляться отдельно, причем для каждого из них надо указывать и тип, и имя. То есть в общем виде список объявлений параметров должен выглядеть следующим образом:

f(тип имяпеременной1, тип имяпеременной2,..., тип имяпеременнойN)

Вот, например, два объявления параметров функций, первое из которых правильное, а второе — нет:

```
f(int i, int k, int j) /* правильное */
f(int i, k, float j)   /* неправильное, у переменной k должен быть
                        собственный спецификатор типа */
```

Что такое область действия функции

В языке правила работы с областями действия — это правила, которые определяют, известен ли фрагменту кода другой фрагмент кода или данных, или имеет ли он доступ к этому другому фрагменту. Об областях действия, определяемых в языке С, говорилось в главе 2. Здесь же мы более подробно рассмотрим одну специальную область действия — ту, которая определяется функцией.

Каждая функция представляет собой конечный блок кода. Таким образом, она определяет область действия этого блока. Это значит, что код функции является закрытым и недоступным ни для какого выражения из любой другой функции, если только не выполняется вызов содержащей его функции. (Например, нельзя перейти в середину другой функции с помощью `goto`.) Код, который составляет тело функции, скрыт от остальной части программы, и если он не использует глобальных переменных, то не может воздействовать на другие части программы или, наоборот, подвергаться воздействию

¹ Данное, возвращаемое функцией, называется также *результатом*. Соответственно, возвращаемый тип часто называется также *типом результата*. — Прим. ред.

с их стороны. Иначе говоря, код и данные, определенные внутри одной функции, без глобальных переменных не могут воздействовать на код и данные внутри другой функции, так как у любых двух разных функций разные области действия.

Переменные, определенные внутри функции, являются локальными. Локальная переменная создается в начале выполнения функции, а при выходе из этой функции она уничтожается. Таким образом, локальная переменная не может сохранять свое значение в промежутках между вызовами функции. Единственное исключение из этого правила — переменные, объявленные со спецификатором класса памяти `static`. Таким переменным память выделяется так же, как и глобальным переменным, которые используются для хранения значений, но область действия таких переменных ограничена содержащими их функциями. (Дополнительная информация о локальных и глобальных переменных приведена в главе 2.)

Формальные параметры функции также находятся в ее области действия. Это значит, что параметр доступен внутри всей функции. Параметр создается в начале выполнения функции, и уничтожается при выходе из нее.

Все функции имеют файл в качестве области действия (file scope). Таким образом, функцию нельзя определять внутри другой функции. Поэтому С практически не является языком с блочной структурой.



Аргументы функции

Если функция должна принимать аргументы, то в ее объявлении следует декларировать параметры, которые примут значения этих аргументов. Как видно из объявления следующей функции, объявления параметров стоят после имени функции.

```
/* Возвращает 1, если символ с входит в строку s, и 0 в противном
   случае. */
int is_in(char *s, char c)
{
    while(*s)
        if(*s==c) return 1;
        else s++;
    return 0;
}
```

Функция `is_in()` имеет два параметра: `s` и `c`. Если символ `c` входит в строку `s`, то эта функция возвращает 1, в противном случае она возвращает 0.

Хотя параметры выполняют специальную задачу, — принимают значения аргументов, передаваемых функции, — они все равно ведут себя так, как и другие локальные переменные. Формальным параметрам функции, например, можно присваивать какие-либо значения или использовать эти параметры в каких-либо выражениях.

Вызовы по значению и по ссылке

В языках программирования имеется два способа передачи значений подпрограмме. Первый из них — *вызов по значению*. При его применении в формальный параметр подпрограммы копируется *значение* аргумента. В таком случае изменения параметра на аргумент не влияют.

Вторым способом передачи аргументов подпрограмме является *вызов по ссылке*. При его применении в параметр копируется адрес аргумента. Это значит, что, в отличие от вызова по значению, изменения значения параметра приводят к точно таким же изменениям значения аргумента.

За небольшим количеством исключений, в языке С для передачи аргументов используется вызов по значению. Обычно это означает, что код, находящийся внутри функции, не может изменять значений аргументов, которые использовались при вызове функции.

Проанализируйте следующую программу:

```
#include <stdio.h>
int sqr(int x);
int main(void)
{
    int t=10;
    printf("%d %d", sqr(t), t);
    return 0;
}

int sqr(int x)
{
    x = x*x;
    return(x);
}
```

В этом примере в параметр *x* копируется 10 — значение аргумента для *sqr()*. Когда выполняется присваивание *x=x*x*, модифицируется только локальная переменная *x*. А значение переменной *t*, использованной в качестве аргумента при вызове *sqr()*, по-прежнему остается равным 10. Поэтому выведено будет следующее: 100 10.

Помните, что именно копия значения аргумента передается в функцию. А то, что происходит внутри функции, не влияет на значение переменной, которая была использована при вызове в качестве аргумента.

Вызов по ссылке

Хотя в С для передачи параметров применяется вызов по значению, можно создать вызов и по ссылке, передавая не сам аргумент, а указатель на него¹. Так как функции передается адрес аргумента, то ее внутренний код в состоянии изменить значение этого аргумента, находящегося, между прочим, за пределами самой функции.

Указатель передается функции так, как и любой другой аргумент. Конечно, в таком случае параметр следует декларировать как один из типов указателей. Это можно увидеть на примере функции *swap()*, которая меняет местами значения двух целых переменных, на которые указывают аргументы этой функции:

```
void swap(int *x, int *y)
{
    int temp;

    temp = *x; /* сохранить значение по адресу x */
    *x = *y;
    *y = temp;
}
```

¹ Конечно, при передаче указателя будет применен вызов по значению, и сам указатель внутри функции вы изменить не сможете. Однако для того объекта, на который указывает этот указатель, все произойдет так, будто этот объект был передан по ссылке. В некоторых языках программирования (например, в Алголе-60) имелись специальные средства, позволяющие уточнить, как следует передавать аргументы: по ссылке или по значению. Благодаря наличию указателей в С механизм передачи параметров удалось унифицировать. Параметры, не являющиеся массивами, в С всегда вызываются только по значению, но все, что в других языках вы можете сделать с объектом, получив ссылку на него (т.е. его адрес), вы можете сделать, получив значение указателя на этот объект (т.е. опять же, его адрес). Так что в языке С благодаря свойственной ему унификации передачи параметров никаких проблем не возникает. А вот в других языках трудности, связанные с отсутствием эффективных средств работы с указателями, встречаются довольно часто. — *Прим. ред.*

```

    *x = *y;    /* поместить y в x */
    *y = temp;  /* поместить x в y */
}

```

Функция `swap()` может выполнять обмен значениями двух переменных, на которые указывают `x` и `y`, потому что передаются их адреса, а не значения. Внутри функции, используя стандартные операции с указателями, можно получить доступ к содержимому переменных и провести обмен их значений¹.

Помните, что `swap()` (или любую другую функцию, в которой используются параметры в виде указателей) необходимо вызывать вместе с *адресами аргументов*². Следующая программа показывает, как надо правильно вызывать `swap()`:

```

#include <stdio.h>
void swap(int *x, int *y);

int main(void)
{
    int i, j;

    i = 10;
    j = 20;

    printf("i и j перед обменом значениями: %d %d\n", i, j);

    swap(&i, &j); /* передать адреса переменных i и j */

    printf("i и j после обмена значениями: %d %d\n", i, j);

    return 0;
}

void swap(int *x, int *y)
{
    int temp;

    temp = *x; /* сохранить значение, хранящееся по адресу x */
    *x = *y;   /* поместить значение y в x */
    *y = temp; /* поместить (старое) значение x в y */
}

```

И вот что вывела эта программа:

```

i и j перед обменом значениями: 10 20
i и j после обмена значениями: 20 10

```

¹ Конечно, задача, решаемая этой программой, кажется тривиальной. Ну разве представляет трудность написать на каком-либо процедурном языке, например, на Алголе-60, процедуру, которая обменивает значения своих параметров. Ведь так просто написать: **procedure** `swap(x, y); integer x, y; begin integer t; t:= x; x:=y; y:=t end`. Но эта процедура работает неправильно, хотя вызов значений здесь происходит по ссылке! Причем сразу найти тестовый пример, демонстрирующий ошибочность этой процедуры, удастся далеко не всем. Ведь в случае вызова `swap(i, j)` все работает правильно! А что будет в случае вызова `swap(i, a[i])`? Да и можно ли на Алголе-60 вообще написать требуемую процедуру? Если вы склоняетесь к отрицательному ответу, то это показывает, насколько все-таки необходимы указатели в развитых языках программирования. Если все же вы *знаете* правильный ответ, то обратите внимание на то, что требуемая процедура, хотя и не длинная, но все же содержит своего рода программистский фокус! — *Прим. ред.*

² Конечно, это просто программистский жаргон. На самом деле, конечно, аргументами являются именно адреса переменных, а не сами переменные. Просто в этом случае для краткости изложения программисты “делают вид”, что вроде бы и в самом деле происходит передача значений *по ссылке*. — *Прим. ред.*

В программе переменной *i* присваивается значение 10, а переменной *j* — значение 20. Затем вызывается функция `swap()` с адресами этих переменных. (Для получения адреса каждой из переменных используется унарный оператор `&`.) Поэтому в `swap()` передаются адреса переменных *i* и *j*, а не их значения.

На заметку

Язык C++ при помощи параметров-ссылок дает возможность полностью автоматизировать вызов по ссылке. А в языке C параметры-ссылки не поддерживаются.

Вызов функций с помощью массивов

Подробно о массивах рассказывалось в главе 4. В настоящем же разделе рассказывается о передаче массивов функциям в качестве аргументов. Этот вопрос рассматривается потому, что эта операция является исключением по отношению к обычной передаче параметров, выполняемой путем вызова по значению¹.

Когда в качестве аргумента функции используется массив, то функции передается его адрес. В этом и состоит исключение по отношению к правилу, которое гласит, что при передаче параметров используется вызов по значению. В случае передачи массива функции ее внутренний код работает с реальным содержимым этого массива и вполне может изменить это содержимое. Проанализируйте, например, функцию `print_upper()`, которая печатает свой строковый аргумент на верхнем регистре:

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

void print_upper(char *string);

int main(void)
{
    char s[80];

    printf("Введите строку символов: ");
    gets(s);
    print_upper(s);
    printf("\ns теперь на верхнем регистре: %s", s);
}

/* Печатать строку на верхнем регистре */
void print_upper(char *string)
{
    register int t;

    for(t=0; string[t]; ++t) {
        string[t] = toupper(string[t]);
        putchar(string[t]);
    }
}
```

Вот что будет выведено в случае фразы “This is a test.” (это тест):

```
Введите строку символов: This is a test.
THIS IS A TEST.
s теперь в верхнем регистре: THIS IS A TEST.
```

(Правда, эта программа не работает с символами кириллицы. — Прим. перев.)

¹ Ведь при вызове по значению пришлось бы копировать весь массив! — Прим. ред.

После вызова `print_upper()` содержимое массива `s` в `main()` переводится в символы верхнего регистра. Если вам это не нужно, программу можно написать следующим образом:

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

void print_upper(char *string);

int main(void)
{
    char s[80];

    printf("Введите строку символов: ");
    gets(s);
    print_upper(s);
    printf("\ns не изменилась: %s", s);

    return 0;
}

void print_upper(char *string)
{
    register int t;

    for(t=0; string[t]; ++t)
        putchar(toupper(string[t]));
}
```

Вот какой на этот раз получится фраза “This is a test.”:

```
Введите строку символов: This is a test.
THIS IS A TEST.
s не изменилась: This is a test.
```

На этот раз содержимое массива не изменилось, потому что внутри `print_upper()` не изменялись его значения.

Классическим примером передачи массивов в функции является стандартная библиотечная функция `gets()`. Хотя `gets()`, которая находится в вашей стандартной библиотеке, и более сложная, чем предлагаемая вам версия `xgets()`, но с помощью функции `xgets()` вы сможете получить представление о том, как работает `gets()`.

```
/* Упрощенная версия стандартной библиотечной функции gets() */
char *xgets(char *s)
{
    char ch, *p;
    int t;

    p = s; /* xgets() возвращает указатель s */

    for(t=0; t<80; ++t){
        ch = getchar();

        switch(ch) {
            case '\n':
                s[t] = '\0'; /* завершить строку */
                return p;
            case '\b':
                if(t>0) t--;
                break;
        }
    }
}
```

```

        default:
            s[t] = ch;
        }
    }
    s[79] = '\0';
    return p;
}

```

Функцию `xgets()` следует вызывать с указателем `char *`. Им, конечно же, может быть имя символьного массива, которое по определению является указателем `char *`. В самом начале программы `xgets()` выполняется цикл `for` от 0 до 80. Это не даст вводить с клавиатуры строки, содержащие более 80 символов. При попытке ввода большего количества символов происходит возврат из функции. (В настоящей функции `gets()` такого ограничения нет.) Так как в языке C нет встроенной проверки границ, программист должен сам позаботиться, чтобы в любом массиве, используемом при вызове `xgets()`, помещалось не менее 80 символов. Когда символы вводятся с клавиатуры, они сразу записываются в строку. Если пользователь нажимает клавишу `<Backspace>`, то счетчик `t` уменьшается на 1, а из массива удаляется последний символ, введенный перед нажатием этой клавиши. Когда пользователь нажмет `<ENTER>`, в конец строки запишется нуль, т.е. признак конца строки. Так как массив, использованный для вызова `xgets()`, модифицируется, то при возврате из функции в нем будут находиться введенные пользователем символы.



Аргументы функции `main()`: `argv` и `argc`

Иногда при запуске программы бывает полезно передать ей какую-либо информацию. Обычно такая информация передается функции `main()` с помощью аргументов командной строки. *Аргумент командной строки* — это информация, которая вводится в командной строке операционной системы вслед за именем программы. Например, чтобы запустить компиляцию программы, необходимо в командной строке после подсказки набрать примерно следующее:

сс имя_программы

имя_программы представляет собой аргумент командной строки, он указывает имя той программы, которую вы собираетесь компилировать.

Чтобы принять аргументы командной строки, используются два специальных встроенных аргумента: `argc` и `argv`. Параметр `argc` содержит количество аргументов в командной строке и является целым числом, причем он всегда не меньше 1, потому что первым аргументом считается имя программы. А параметр `argv` является указателем на массив указателей на строки. В этом массиве каждый элемент указывает на какой-либо аргумент командной строки. Все аргументы командной строки являются строковыми, поэтому преобразование каких бы то ни было чисел в нужный двоичный формат должно быть предусмотрено в программе при ее разработке.

Вот простой пример использования аргумента командной строки. На экран выводятся слово *Привет* и ваше имя, которое надо указать в виде аргумента командной строки.

```

#include <stdio.h>
#include <stdlib.h>
int main(int argc, char *argv[])
{
    if(argc!=2) {
        printf("Вы забыли ввести свое имя.\n");
        exit(1);
    }
}

```

```
printf("Привет, %s", argv[1]);

return 0;
}
```

Если вы назвали эту программу `name` (имя) и ваше имя Том, то для запуска программы следует в командную строку ввести `name Том`. В результате выполнения программы на экране появится сообщение `Привет, Том`.

Во многих средах все аргументы командной строки необходимо отделять друг от друга пробелом или табуляцией. Запятые, точки с запятой и тому подобные символы разделителями не считаются. Например,

```
run Spot, run
```

состоит из трех символьных строк, в то время как

```
Эрб, Рик, Фред
```

представляет собой одну символьную строку — запятые, как правило, разделителями не считаются.

Если в строке имеются пробелы, то, чтобы из нее не получилось несколько аргументов, в некоторых средах эту строку можно заключать в двойные кавычки. В результате вся строка будет считаться одним аргументом. Чтобы подробнее узнать, как в вашей операционной системе задаются параметры командной строки, изучите документацию этой системы.

Очень важно правильно объявлять `argv`. Вот как это делают чаще всего:

```
char *argv[];
```

Пустые квадратные скобки указывают на то, что у массива неопределенная длина. Теперь получить доступ к отдельным аргументам можно с помощью индексации массива `argv`. Например, `argv[0]` указывает на первую символьную строку, которой всегда является имя программы; `argv[1]` указывает на первый аргумент и так далее.

Другим небольшим примером использования аргументов командной строки является приведенная далее программа `countdown` (счет в обратном порядке). Эта программа считает в обратном порядке, начиная с какого-либо значения (указанного в командной строке), и подает звуковой сигнал, когда доходит до 0. Обратите внимание, что первый аргумент, содержащий начальное значение, преобразуется в целое значение с помощью стандартной функции `atoi()`. Если вторым аргументом командной строки (а если считать аргументом имя программы, то третьим — *Прим. перев.*) является строка `"display"` (вывод на экран), то результат отсчета (в обратном порядке) будет выводиться на экран.

```
/* Программа счета в обратном порядке */
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <string.h>

int main(int argc, char *argv[])
{
    int disp, count;

    if(argc<2) {
        printf("В командной строке необходимо ввести число,\n"
               с которого\n");
        printf("начинается отсчет. Попробуйте снова.\n");
        exit(1);
    }
}
```

```

if(argc==3 && !strcmp(argv[2], "display")) disp = 1;
else disp = 0;

for(count=atoi(argv[1]); count; --count)
    if(disp) printf("%d\n", count);

putchar('\a'); /* здесь подается звуковой сигнал */
printf("Счет закончен");

return 0;
}

```

Обратите внимание, если аргументы командной строки не будут указаны, то будет выведено сообщение об ошибке. В программах с аргументами командной строки часто делается следующее: в случае, когда пользователь запускает эти программы без ввода нужной информации, выводятся инструкции о том, как правильно указывать аргументы.

Чтобы получить доступ к отдельному символу одного из аргументов командной строки, введите в `argv` второй индекс. Например, следующая программа посимвольно выводит все аргументы, с которыми ее вызвали:

```

#include <stdio.h>

int main(int argc, char *argv[])
{
    int t, i;
    for(t=0; t<argc; ++t) {
        i = 0;

        while(argv[t][i]) {
            putchar(argv[t][i]);
            ++i;
        }
        printf("\n");
    }

    return 0;
}

```

Помните, первый индекс `argv` обеспечивает доступ к строке, а второй индекс — доступ к ее отдельным символам.

Обычно `argc` и `argv` используют для того, чтобы передать программе начальные команды, которые понадобятся ей при запуске. Например, аргументы командной строки часто указывают такие данные, как имя файла, параметр или альтернативное поведение. Использование аргументов командной строки придает вашей программе “профессиональный внешний вид” и облегчает ее использование в пакетных файлах.

Имена `argc` и `argv` являются традиционными, но не обязательными. Эти два параметра в функции `main()` вы можете назвать как угодно. Кроме того, в некоторых компиляторах для `main()` могут поддерживаться дополнительные аргументы, поэтому обязательно изучите документацию к вашему компилятору.

Когда для программы не требуются параметры командной строки, то чаще всего явно декларируют функцию `main()` как не имеющую параметров. В таком случае в списке параметров этой функции используют ключевое слово `void`.



Оператор return

Механизм использования `return` описан в главе 3. Как вы помните, там говорится, что этот оператор имеет два важных применения. Во-первых, он обеспечивает немедленный выход из функции, т.е. заставляет выполняющуюся программу передать управление коду, вызвавшему функцию. Во-вторых, этот оператор можно использовать для того, чтобы возвратить значение. В следующих разделах рассказывается, каким именно образом можно использовать оператор `return`.

Возврат из функции

Функция может завершать выполнение и осуществлять возврат в вызывающую программу двумя способами. Первый способ используется тогда, когда после выполнения последнего оператора в функции встречается закрывающая фигурная скобка `()`. (Конечно, это просто жаргон, ведь в настоящем объектном коде фигурной скобки нет!) Например, функция `pr_reverse()` в приведенной ниже программе просто выводит на экран в обратном порядке строку `Мне нравится С`, а затем возвращает управление вызывающей программе.

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

void pr_reverse(char *s);

int main(void)
{
    pr_reverse("Мне нравится С");

    return 0;
}

void pr_reverse(char *s)
{
    register int t;

    for(t=strlen(s)-1; t>=0; t--) putchar(s[t]);
}
```

Как только строка выведена на экран, функции `pr_reverse()` “делать больше нечего”, поэтому она возвращает управление туда, откуда она была вызвана.

Но на практике не так уж много функций используют именно такой способ завершения выполнения. В большинстве функций для завершения выполнения используется оператор `return` — или потому, что необходимо вернуть значение, или чтобы сделать код функции проще и эффективнее.

В функции может быть несколько операторов `return`. Например, в следующей программе функция `find_substr()` возвращает начальную позицию подстроки в строке или же возвращает `-1`, если подстрока, наоборот, не найдена. В этой функции для упрощения кодирования используются два оператора `return`.

```
#include <stdio.h>

int find_substr(char *s1, char *s2);

int main(void)
{
```



```

    if(find_substr("С — это забавно", "это") != -1)
        printf("Подстрока найдена.");

    return 0;
}

/* Вернуть позицию первого вхождения s2 в s1. */
int find_substr(char *s1, char *s2)
{
    register int t;
    char *p, *p2;

    for(t=0; s1[t]; t++) {
        p = &s1[t];
        p2 = s2;

        while(*p2 && *p2==*p) {
            p++;
            p2++;
        }
        if(!*p2) return t; /* 1-й оператор return */
    }
    return -1; /* 2-й оператор return */
}

```

Возврат значений

Все функции, кроме тех, которые относятся к типу `void`, возвращают значение. Это значение указывается выражением в операторе `return`. Стандарт C89 допускает выполнение оператора `return` без указания выражения внутри функции, тип которой отличен от `void`. В этом случае все равно происходит возврат какого-нибудь произвольного значения. Но такое положение дел, мягко говоря, никуда не годится! Поэтому в Стандарте C99 (да и в C++) предусмотрено, что в функции, тип которой отличен от `void`, в операторе `return` необходимо *обязательно* указать возвращаемое значение. То есть, согласно C99, если для какой-либо функции указано, что она возвращает значение, то внутри этой функции у любого оператора `return` должно быть свое выражение. Однако если функция, тип которой отличен от `void`, выполняется до самого конца (то есть до закрывающей ее фигурной скобки), то возвращается произвольное (непредсказуемое с точки зрения разработчика программы!) значение. Хотя здесь нет синтаксической ошибки, это является серьезным упущением и таких ситуаций необходимо избегать.

Если функция не объявлена как имеющая тип `void`, она может использоваться как операнд в выражении. Поэтому каждое из следующих выражений является правильным:

```

x = power(y);
if(max(x,y) > 100) printf("больше");
for(ch=getchar(); isdigit(ch); ) ...;

```

Общепринятое правило гласит, что вызов функции не может находиться в левой части оператора присваивания. Выражение

```

swap(x,y) = 100; /* неправильное выражение */

```

является неправильным. Если компилятор C в какой-либо программе найдет такое выражение, то пометит его как ошибочное и программу компилировать не будет.

В программе можно использовать функции трех видов. Первый вид — простые вычисления. Эти функции предназначены для выполнения операций над своими аргументами и возвращают полученное в результате этих операций значение. Вычисли-

тельная функция является функцией “в чистом виде”. В качестве примеров можно назвать стандартные библиотечные функции `sqrt()` и `sin()`, которые вычисляют квадратный корень и синус своего аргумента соответственно.

Второй вид включает в себя функции, которые обрабатывают информацию и возвращают значение, которое показывает, успешно ли была выполнена эта обработка. Примером является библиотечная функция `fclose()`, которая закрывает файл. Если операция закрытия была завершена успешно, функция возвращает 0, а в случае ошибки она возвращает EOF.

У функций последнего, третьего вида нет явно возвращаемых значений. В сущности, такие функции являются чисто процедурными и никаких значений выдавать не должны. Примером является `exit()`, которая прекращает выполнение программы. Все функции, которые не возвращают значение, должны объявляться как возвращающие значение типа `void`. Объявляя функцию как возвращающую значение типа `void`, вы запрещаете ее применение в выражениях, предотвращая таким образом случайное использование этой функции не по назначению.

Иногда функции, которые, казалось бы, фактически не выдают содержательный результат, все же возвращают какое-то значение. Например, `printf()` возвращает количество выведенных символов. Если бы нашлась такая программа, которая на самом деле проверяла бы это значение, то это было бы что-то необычное... Другими словами, хотя все функции, за исключением относящихся к типу `void`, возвращают значения, вовсе не нужно стремиться использовать эти значения во что бы то ни стало. Часто при обсуждении значений, возвращаемых функциями, возникает такой довольно распространенный вопрос: “Неужели не обязательно присваивать возвращенное значение какой-либо переменной? Не повиснет ли оно где-нибудь и не приведет ли это в дальнейшем к каким-либо неприятностям?” Отвечая на этот вопрос, повторим, что присваивание отнюдь не является обязательным, причем отсутствие его не станет причиной каких-либо неприятностей. Если возвращаемое значение не входит ни в один из операторов присваивания, то это значение будет просто отброшено. Отметим также, что такое отбрасывание значения встречается очень часто. Проанализируйте следующую программу, в которой используется функция `mul()`:

```
#include <stdio.h>

int mul(int a, int b);

int main(void)
{
    int x, y, z;

    x = 10;  y = 20;
    z = mul(x,y);          /* 1 */
    printf("%d", mul(x,y)); /* 2 */
    mul(x,y);              /* 3 */

    return 0;
}

int mul(int a, int b)
{
    return a*b;
}
```

В строке 1 значение, возвращаемое функцией `mul()`, присваивается переменной `z`. В строке 2 возвращаемое значение не присваивается, но используется функцией `printf()`. И наконец, в строке 3 возвращаемое значение теряется, потому что не присваивается никакой из переменных и не используется как часть какого-либо выражения.

Возвращаемые указатели

Хотя с функциями, которые возвращают указатели, обращаются так же, как и с любыми другими функциями, все же будет полезно познакомиться с некоторыми основными понятиями и рассмотреть соответствующий пример. Указатели не являются ни целыми, ни целыми без знака. Они являются адресами в памяти и относятся к особому типу данных. Такая особенность указателей определяется тем, что арифметика указателей (адресная арифметика) работает с учетом параметров базового типа. Например, если указателю на целое придать минимальное (ненулевое) приращение, то его текущее значение станет на четыре больше, чем предыдущее (при условии, что целые значения занимают 4 байта). Вообще говоря, каждый раз, когда значение указателя увеличивается (уменьшается) на минимальную величину, то он указывает на последующий (предыдущий) элемент, имеющий базовый тип указателя. Так как размеры разных типов данных могут быть разными, то компилятор должен знать тип данных, на которые может указывать указатель. Поэтому в объявлении функции, которая возвращает указатель, тип возвращаемого указателя должен декларироваться явно. Например, нельзя объявлять возвращаемый тип как `int *`, если возвращается указатель типа `char *`! Иногда (правда, крайне редко!) требуется, чтобы функция возвращала “универсальный” указатель, т.е. указатель, который может указывать на данные любого типа. Тогда тип результата функции следует определить как `void *`.

Чтобы функция могла вернуть указатель, она должна быть объявлена как возвращающая указатель на нужный тип. Например, следующая функция возвращает указатель на первое вхождение символа, присвоенного переменной `c`, в строку `s`. Если этого символа в строке нет, то возвращается указатель на символ конца строки (`'0'`).

```
/* Возвращает указатель на первое вхождение c в s. */
char *match(char c, char *s)
{
    while(c!=*s && *s) s++;
    return(s);
}
```

Вот небольшая программа, в которой используется функция `match()`:

```
#include <stdio.h>

char *match(char c, char *s); /* прототип */

int main(void)
{
    char s[80], *p, ch;

    gets(s);
    ch = getchar();
    p = match(ch, s);

    if(*p) /* символ найден */
        printf("%s ", p);
    else
        printf("Символа нет.");

    return 0;
}
```

Эта программа сначала считывает строку, а затем символ. Потом проводится поиск местонахождения символа в строке. При наличии символа в строке переменная `p` укажет на него, и программа выведет строку, начиная с найденного символа. Если символ в строке не найден, то `p` укажет на символ конца строки (`'0'`), причем `*p` будет представлять логическое значение ЛОЖЬ (`false`). В таком случае программа выведет сообщение Символа нет.

Функции типа void

Одним из применений ключевого слова `void` является явное объявление функций, которые не возвращают значений. Мы уже знаем, что такие функции не могут применяться в выражениях, и указание ключевого слова `void` предотвращает их случайное использование не по назначению. Например, функция `print_vertical()` выводит в боковой части экрана свой строчный аргумент по вертикали сверху вниз.

```
void print_vertical(char *str)
{
    while(*str)
        printf("%c\n", *str++);
}
```

Вот пример использования функции `print_vertical()`:

```
#include <stdio.h>

void print_vertical(char *str); /* прототип */

int main(int argc, char *argv[])
{
    if(argc > 1) print_vertical(argv[1]);

    return 0;
}

void print_vertical(char *str)
{
    while(*str)
        printf("%c\n", *str++);
}
```

И еще одно замечание: в ранних версиях C ключевое слово `void` не определялось. Таким образом, в программах, написанных на этих версиях C, функции, которые не возвращали значений, просто имели по умолчанию тип `int` — и это несмотря на то, что они не возвращали никаких значений!

Что возвращает функция `main()`?

Функция `main()` возвращает целое число, которое принимает вызывающий процесс — обычно этим процессом является операционная система. Возврат значения из `main()` эквивалентен вызову функции `exit()` с тем же самым значением. Если `main()` не возвращает значение явно, то вызывающий процесс получает формально неопределенное значение. На практике же большинство компиляторов C автоматически возвращают 0, но если встает вопрос переносимости, то на такой результат полагаться с уверенностью нельзя.

Рекурсия

В языке C функция может вызывать сама себя. В этом случае такая функция называется *рекурсивной*. Рекурсия — это процесс определения чего-либо на основе самого себя, из-за чего рекурсию еще называют *рекурсивным определением*.

Простым примером рекурсивной функции является `factr()`, которая вычисляет факториал целого неотрицательного числа. Факториалом числа n (обозначается $n!$ — *Прим. перев.*) называется произведение всех целых чисел, от 1 до n включительно (для 0, по определению, факториал равен 1. — *Прим. перев.*). Например, $3!$ — это $1 \times 2 \times 3$, или 6. Здесь показаны `factr()` и эквивалентная ей функция, в которой используется итерация:

```
/* рекурсивная функция */
int factr(int n) {
    int answer;

    if(n==0) return(1);
    answer = factr(n-1)*n; /* рекурсивный вызов */
    return(answer);
}

/* нерекурсивная функция */
int fact(int n) {
    int t, answer;

    answer = 1;
    for(t=1; t<=n; t++)
        answer=answer*(t);

    return(answer);
}
```

Нерекурсивное вычисление факториала, то есть вычисление с помощью `fact()`, выполняется достаточно просто. В этой функции в теле цикла, выполняющемся для t от 1 до n , вычисленное ранее произведение последовательно умножается на каждое из этих чисел. (Значение факториала для 0 получается, конечно, с помощью оператора присваивания. Значение факториала для 1 также получается умножением не на ранее полученное произведение, а на заранее подготовленное число, тоже равное 1. — *Прим. перев.*)

Работа же рекурсивной функции `factr()` чуть более сложная. Когда `factr()` вызывается с аргументом 0, то она сразу возвращает 1. Если же аргумент больше 0, то возвращается произведение `factr(n-1)*n`. Чтобы вычислить значение этого выражения, `factr()` вызывается с аргументом $n-1$. Это выполняется до тех пор, пока n не станет равным 0. Когда это произойдет, вызовы функции начнут возвращать вычисленные ими значения факториалов.

При вычислении $2!$ первый вызов `factr()` влечет за собой второй, теперь уже рекурсивный вызов с аргументом 1, который, в свою очередь, влечет третий, тоже рекурсивный вызов с аргументом 0. Этот вызов возвращает число 1, которое затем умножается на 1, а потом на 2 (первоначальное значение n). Ответ в данном случае равен 2. Попробуйте самостоятельно вычислить $3!$. (Вам, возможно, захочется вставить в функцию `factr()` выражения `printf()`, чтобы видеть уровень каждого вывода, и то, какие будут промежуточные ответы.)

Когда функция вызывает сама себя, новый набор локальных переменных и параметров размещается в памяти в стеке, а код функции выполняется с самого своего начала, причем используются именно эти новые переменные. При рекурсивном вызове функции новая копия ее кода не создается. Новыми являются только значения, которые использует данная функция. При каждом возвращении из рекурсивного вызова старые локальные переменные и параметры извлекаются из стека, и сразу за рекурсивным вызовом возобновляется работа функции. При использовании рекурсивных функций стек работает подобно “телескопической” трубе, выдвигающейся вперед и складывающейся обратно.

Хотя и кажется, что рекурсия предлагает более высокую эффективность, но на самом деле такое бывает достаточно редко. Использование рекурсии в программах зачастую не очень сильно уменьшают их размер кода и обычно только незначительно увеличивает эффективность использования памяти. Кроме того, рекурсивные версии большинства программ могут выполняться несколько медленнее, чем их итеративные варианты, потому что при рекурсивных вызовах функций расходуются дополнительные ресурсы. Кроме того, большое количество рекурсивных вызовов функции может вызвать переполнение стека. Из-за того, что память для параметров функции и локальных переменных находится в стеке и при каждом новом вызове создается еще один набор этих переменных, то для переменных места в стеке может рано или поздно не хватить. Переполнение стека — вот обычная причина аварийного завершения программы, когда функция утрачивает контроль над рекурсивными обращениями.

Главным преимуществом рекурсивных функций является то, что с их помощью упрощается реализация некоторых алгоритмов, а программа становится понятнее. Например, алгоритм быстрой сортировки (описанный в части IV) трудно реализовать итеративным способом. Кроме того, для некоторых проблем, особенно связанных с искусственным интеллектом, больше подходят рекурсивные решения. И наконец, некоторым людям легче думать рекурсивными категориями, чем итеративными.

В тексте рекурсивной функции обязательно должен быть выполнен условный оператор, например `if`, который при определенных условиях вызовет завершение функции, т.е. возврат, а не выполнит очередной рекурсивный вызов. Если такого оператора нет, то после вызова функция никогда не сможет завершить работы. Распространенной ошибкой при написании рекурсивных функций как раз и является отсутствие в них условного оператора. При создании программ не отказывайтесь от функции `printf()`; тогда вы сможете увидеть, что происходит на самом деле и сможете прервать выполнение, когда обнаружите ошибку.

Прототипы функций

В современных, правильно написанных программах на языке C каждую функцию перед использованием необходимо объявлять. Обычно это делается с помощью *прототипа функции*. В первоначальном варианте языка C прототипов не было; но они были введены уже в Стандарт C89. Хотя прототипы формально не требуются, но их использование очень желательно. (Впрочем, в C++ прототипы *обязательны*!) Во всех примерах этой книги имеются полные прототипы функций. Прототипы дают компилятору возможность тщательнее выполнять проверку типов, подобно тому, как это делается в таких языках как Pascal. Если используются прототипы, то компилятор может обнаружить любые сомнительные преобразования типов аргументов, необходимые при вызове функции, если тип ее параметров отличается от типов аргументов. При этом будут выданы предупреждения обо всех таких сомнительных преобразованиях. Компилятор также обнаружит различия в количестве аргументов, использованных при вызове функции, и в количестве параметров функции.

В общем виде прототип функции должен выглядеть таким образом:

тип имя_функции(тип имя_парам1, тип имя_парам2, ..., имя_парамN);

Использование имен параметров не обязательно. Однако они дают возможность компилятору при наличии ошибки указать имена, для которых обнаружено несоответствие типов, так что не поленитесь указать этих имен — это позволит сэкономить время впоследствии.

Следующая программа показывает, насколько ценными являются прототипы функций. В ней выводится сообщение об ошибке, происходящей из-за того, что программа содержит попытку вызова `sqr_it()` с целым аргументом, в то время как требуется указатель на целое.

```

/* В этой программе используется прототип функции,
чтобы обеспечить тщательную проверку типов. */

void sqr_it(int *i); /* прототип */

int main(void)
{
    int x;

    x = 10;
    sqr_it(x); /* несоответствие типов */

    return 0;
}

void sqr_it(int *i)
{
    *i = *i * *i;
}

```

В качестве прототипа функции может также служить ее определение, если оно находится в программе до первого вызова этой функции. Вот, например, правильная программа:

```

#include <stdio.h>

/* Это определение будет также служить и
прототипом внутри этой программы. */
void f(int a, int b)
{
    printf("%d ", a % b);
}

int main(void)
{
    f(10,3);

    return 0;
}

```

В этом примере специальный прототип не требуется; так как функция `f()` определена еще до того, как она начинает использоваться в `main()`. Хотя определение функции и может служить ее прототипом в малых программах, но в больших такое встречается редко — особенно, когда используется несколько файлов. В программах, приведенных в качестве примеров в этой книге, для каждой функции автор старался приводить отдельный прототип потому, что именно так обычно и пишется код на языке С.

Единственная функция, для которой не требуется прототип — это `main()`, так как это первая функция, вызываемая в начале работы программы.

Имеется небольшая, но важная разница в том, как именно в С и С++ обрабатывается прототип функции, не имеющей параметров. В С++ пустой список параметров указывается полным отсутствием в прототипе любых параметров. Например,

```

int f(); /* Прототип в С++ для функции, не имеющей параметров */

```

Однако в С это выражение означает нечто другое. Из-за необходимости придерживаться совместимости с первоначальной версией С пустой список параметров сообщает, что просто *о параметрах* не предоставлено *никакой информации*. Что касается компилятора, то для него эта функция может иметь несколько параметров, а может не иметь ни одного. (Такой оператор называется старомодным объявлением функции, он описан в следующем разделе.)

Если функция в языке C не имеет параметров, то в ее прототипе внутри списка параметров стоит только ключевое слово `void`. Вот, например, прототип функции `f()` в том виде, в каком он должен быть в программе на языке C:

```
float f(void);
```

Таким образом компилятор узнает, что у функции нет параметров, и любое обращение к ней, в котором имеются аргументы, будет считаться ошибкой. В C++ использование ключевого слова `void` внутри пустого списка параметров также разрешено, но считается излишним.

Прототипы функций позволяют “отлавливать” ошибки еще до запуска программы. Кроме того, они запрещают вызов функций при несовпадении типов (т.е. с неподходящими аргументами) и тем самым помогают проверять правильность программы.

И напоследок хотелось бы сказать следующее: так как в ранних версиях C синтаксис прототипов в полном объеме не поддерживался, то в C прототипы формально не обязательны. Такой подход необходим для совместимости с C-кодом, созданным еще до появления прототипов. Но если старый C-код переносится в C++, то перед компиляцией этого кода в него необходимо добавить полные прототипы функций. Помните, что хотя прототипы в C не обязательны, но они обязательны в C++. Это значит, что каждая функция в программе на языке C++ должна иметь полный прототип. Поэтому при написании программ на C в них указываются полные прототипы функций — именно так поступает большинство программистов, работающих на этом языке.

Старомодные объявления функций

В “ранней молодости” языка C, еще до создания прототипов функций, все-таки была необходимость сообщить компилятору о типе результата функции, чтобы при вызове функции был создан правильный код. (Так как размеры разных типов данных разные, то размер типа результата надо было знать еще до вызова функции.) Это выполнялось с помощью объявления функции, не содержащего никакой информации о параметрах. С точки зрения теперешних стандартов этот старомодный подход является архаичным. Однако его до сих пор можно найти в старых кодах. По этой причине важно понимать, как он работает.

Согласно старомодному подходу, тип результата и имя функции, как показано ниже, объявляются почти что в начале программы:

```
#include <stdio.h>

double div(); /* старомодное объявление функции */
int main(void)
{
    printf("%f", div(10.2, 20.0));

    return 0;
}

double div(double num, double denom)
{
    return num / denom;
}
```

Старомодное объявление типа функции сообщает компилятору, что функция `div()` возвращает результат типа `double`. Это объявление позволяет компилятору правильно генерировать код для вызовов этой функции. Однако оно ничего не говорит о параметрах `div()`.

Общий вид старомодного оператора объявления функции такой:

спецификатор_типа имя_функции();

Обратите внимание, что список параметров пустой. Даже если функция принимает аргументы, то ни один из них не перечисляется в объявлении типа.

Как уже говорилось, старомодное объявление функции устарело и не должно использоваться в новом коде. Кроме того, оно несовместимо с C++.

Прототипы стандартных библиотечных функций

Любая стандартная библиотечная функция в программе должна иметь прототип. Поэтому для каждой такой функции необходимо ввести соответствующий *заголовок*. Все необходимые заголовки предоставляются компилятором C. В системе программирования на языке C библиотечными заголовками (обычно) являются файлы, в именах которых используется расширение `.h`. В заголовке имеется два основных элемента: любые определения, используемые библиотечными функциями, и прототипы библиотечных функций. Например, почти во все программы из этой книги включается файл `<stdio.h>`, потому что в этом файле находится прототип для `printf()`. Заголовки для стандартных функций описаны в части II.

Объявление списков параметров переменной длины

Можно вызвать функцию, которая имеет переменное количество параметров. Самым известным примером является `printf()`. Чтобы сообщить компилятору, что функции будет передано заранее неизвестное количество аргументов, объявление списка ее параметров необходимо закончить многоточием. Например, следующий прототип указывает, что у функции `func()` будет как минимум два целых параметра и после них еще некоторое количество (в том числе и 0) параметров:

```
int func(int a, int b, ...);
```

В любой функции, использующей переменное количество параметров, должен быть как минимум один реально существующий параметр. Например, следующее объявление неправильное:

```
int func(...); /* ошибка */
```

Правило “неявного int”

Первоначальная версия C отличалась особенностью, которую иногда называют правилом “неявного int” (а также правилом “int по умолчанию”). Это правило состоит в том, что если спецификатор базового типа явно не указан, то подразумевается спецификатор `int`. Это правило было включено в стандарт C89, но в C99 это правило не вошло. (И, кроме того, не поддерживается в языке C++.) Так как правило “неявного int” теперь устарело, то в этой книге оно не используется. Однако из-за того, что во многих действующих программах это правило еще используется, то о нем следует немного рассказать.

Больше всего правило “неявного int” использовалось при определении типа результата функции, т.е. при определении возвращаемого типа. Много лет назад большинство программистов, писавших программы на C, пользовались этим правилом, когда писали код функций, возвращавших результат типа `int`. Поэтому много лет назад такая функция, как

```
int f(void) {
    /* ... */
    return 0;
}
```

часто могла быть написана таким образом:

```
f(void) { /* int — возвращаемый тип по умолчанию */
    /* ... */
    return 0;
}
```

В первом случае возвращаемый тип `int` определяется явно. Во втором же — подразумевается по умолчанию.

Правило “ неявного `int` ” применяется не только к значениям, возвращаемым функциями (хотя это было самое распространенное применение). Например, для C89 и более ранних вариантов C правильной является следующая функция:

```
/* по умолчанию возвращается тип int,
   такой же тип, как и у параметров a и b */
f(register a, register b) {
    register c; /* переменная c по умолчанию также будет иметь тип int */

    c = a + b;

    printf("%d", c);

    return c;
}
```

Здесь возвращаемым типом по умолчанию для `f()` является `int`; т.е. такой же тип по умолчанию, как и у параметров `a` и `b`, и у локальной переменной `c`.

Помните, что правило “ неявного `int` ” не поддерживается в C99 или C++. Таким образом, использовать его в программах, совместимых с C89, не рекомендуется. Лучше всего явно определять каждый тип, используемый в вашей программе.

Старомодные и современные объявления параметров функций

В ранних версиях C использовался синтаксис объявления параметров, отличающийся от того, который используется в современных версиях этого языка, включая C89, C99 и C++. Такой ранний синтаксис иногда называется *классическим*. В отличие от него, синтаксис, который используется в этой книге, называется *современным*. В стандартном C поддерживаются оба синтаксиса, но настоятельно рекомендуется современный. (А в C++ поддерживается только современный синтаксис объявления параметров.) Однако старомодный синтаксис надо знать, потому что он до сих пор используется во многих старых программах, написанных на языке C.

Старомодное объявление параметров функции состоит из двух частей: списка параметров внутри круглых скобок, следующих за именем функции, а также объявлений параметров, находящихся между закрывающей круглой скобкой и открывающей фигурной скобкой функции. В общем виде старомодное определение параметров должно выглядеть таким образом:

```
тип имя_функции(парм1, парм2,...пармN)
тип парм1;
```

```

тип парм2;
.
.
.
тип пармN;
{
код функции
}

```

Например, такое современное объявление, как

```

float f(int a, int b, char ch)
{
    /* ... */
}

```

в старомодном виде будет выглядеть следующим образом:

```

float f(a, b, ch)
int a, b;
char ch;
{
    /* ... */
}

```

Обратите внимание, что старомодный синтаксис позволяет в списке, стоящем за именем типа, объявить более одного параметра.

На заметку

Старомодный синтаксис объявления параметров признан устаревшим для стандартного C и не поддерживается в языке C++.

Ключевое слово inline

В Стандарте C99 было введено ключевое слово `inline`, применяемое к функциям. Оно подробно рассматривается в части II, а здесь мы дадим только его краткое описание. Ставя перед объявлением функции ключевое слово `inline`, вы даете компилятору указание оптимизировать вызовы этой функции. Обычно это означает, что такие вызовы желательно заменить последовательной вставкой кода самой функции. Однако `inline` является всего лишь запросом к компилятору и может быть проигнорировано.

На заметку

Спецификатор `inline` также поддерживается в языке C++.

Полный
справочник по



Глава 7

**Структуры, объединения,
перечисления и декларация
`typedef`**

В языке C имеется пять способов создания пользовательских типов данных. Пользовательские типы данных можно создать с помощью:

- *структуры* — группы переменных, имеющей одно имя и называемой *агрегатным* типом данных. (Кроме того, еще известны термины *соединение* (*compound*) и *конгломерат* (*conglomerate*).);
- *объединения*, которое позволяет определять один и тот же участок памяти как два или более типов переменных;
- *битового поля*, которое является специальным типом элемента структуры или объединения, позволяющим легко получать доступ к отдельным битам;
- *перечисления* — списка поименованных целых констант;
- ключевого слова `typedef`, которое определяет новое имя для существующего типа.

Все эти способы описаны в этой главе.



Структуры

Структура — это совокупность переменных, объединенных под одним именем. С помощью структур удобно размещать в смежных полях связанные между собой элементы информации. *Объявление структуры* создает шаблон, который можно использовать для создания ее объектов (то есть экземпляров этой структуры). Переменные, из которых состоит структура, называются *членами*. (Члены структуры еще называются *элементами* или *полями*.)

Как правило, члены структуры связаны друг с другом по смыслу. Например, элемент списка рассылки, состоящий из имени и адреса логично представить в виде структуры. В следующем фрагменте кода показано, как объявить структуру, в которой определены поля имени и адреса. Ключевое слово `struct` сообщает компилятору, что объявляется (еще говорят, “декларируется”) структура.

```
struct addr
{
    char name[30];
    char street[40];
    char city[20];
    char state[3];
    unsigned long int zip;
};
```

Обратите внимание, что объявление завершается точкой с запятой, потому что объявление структуры является оператором. Кроме того, *тег* структуры `addr` идентифицирует эту конкретную структуру данных и является спецификатором ее типа.

В данном случае *на самом деле никакая переменная не создается*. Всего лишь определяется вид данных. Когда вы объявляете структуру, то определяете агрегатный *тип*, а не переменную. И пока вы не объявите переменную этого типа, то существовать она не будет. Чтобы объявить переменную (то есть физический объект) типа `addr`, напишите следующее:

```
struct addr addr_info;
```

В этом операторе объявлена переменная типа `addr`, которая называется `addr_info`. Таким образом, `addr` описывает вид структуры (ее тип), а `addr_info` является экземпляром (объектом) этой структуры.

Когда объявляется переменная-структура, компилятор автоматически выделяет количество памяти, достаточное, чтобы разместить все ее члены. На рис. 7.1 показано, как `addr_info` размещена в памяти; в данном случае предполагается, что целые переменные типа `long` занимают по 4 байта.

Одновременно с объявлением структуры можно объявить одну или несколько переменных. Например,

```
struct addr {  
    char name[30];  
    char street[40];  
    char city[20];  
    char state[3];  
    unsigned long int zip;  
} addr_info, binfo, cinfo;
```

определяет тип структуры, называемый `addr`, и объявляет переменные этого типа `addr_info`, `binfo` и `cinfo`. Важно понимать, что каждая переменная-структура содержит собственные копии членов структуры. Например, поле `zip` в `binfo` отличается от поля `zip` в `cinfo`. Изменения в `zip` из `binfo` не повлияют на содержимое поля `zip`, находящегося в `cinfo`.

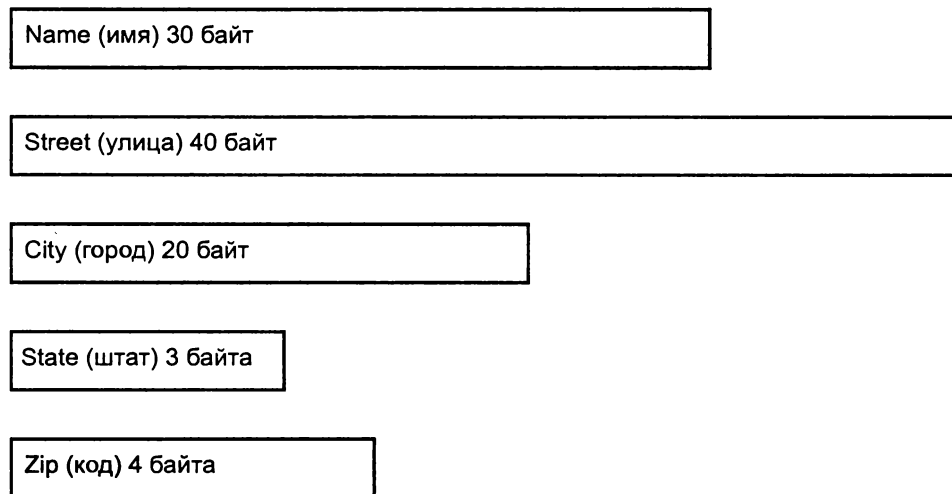


Рис. 7.1. Расположение в памяти структуры `addr_info`

Если нужна только одна переменная-структура, то *тег* структуры является лишним. В этом случае наш пример объявления можно переписать следующим образом:

```
struct {  
    char name[30];  
    char street[40];  
    char city[20];  
    char state[3];  
    unsigned long int zip;  
}
```

В этом случае объявляется одна переменная с именем `addr_info`, причем ее поля указаны в структуре, которая предшествует этому имени.

Общий вид объявления структуры такой:

```

struct me2 {
    тип имя-члена;
    тип имя-члена;
    тип имя-члена;
    .
    .
    .
} переменные-структуры;

```

причем *тег* или *переменные-структуры* могут быть пропущены, но только не оба одновременно.

Доступ к членам структуры

Доступ к отдельным членам структуры осуществляется с помощью оператора `.` (который обычно называют *оператором точка* или *оператором доступа к члену структуры*). Например, в следующем выражении полю `zip` в уже объявленной переменной-структуре `addr_info` присваивается значение ZIP-кода, равное 12345:

```

addr_info.zip = 12345;

```

Этот отдельный член определяется именем объекта (в данном случае `addr_info`), за которым следует точка, а затем именем самого этого члена (в данном случае `zip`). В общем виде использование оператора точка для доступа к члену структуры выглядит таким образом:

имя-объекта.имя-члена

Поэтому, чтобы вывести ZIP-код на экран, напишите следующее:

```

printf("%lu", addr_info.zip);

```

Будет выведен ZIP-код, который находится в члене `zip` переменной-структуры `addr_info`.

Точно так же в вызове `gets()` можно использовать массив символов `addr_info.name`:

```

gets(addr_info.name);

```

Таким образом, в начало `name` передается указатель на символьную строку.

Так как `name` является массивом символов, то чтобы получить доступ к отдельным символам в массиве `addr_info.name`, можно использовать индексы вместе с `name`. Например, с помощью следующего кода можно посимвольно вывести на экран содержимое `addr_info.name`:

```

for(t=0; addr_info.name[t]; ++t)
    putchar(addr_info.name[t]);

```

Обратите внимание, что индексируется именно `name` (а не `addr_info`). Помните, что `addr_info` — это имя всего объекта-структуры, а `name` — имя элемента этой структуры. Таким образом, если требуется индексировать элемент структуры, то индекс необходимо указывать после имени этого элемента.

Присваивание структур

Информация, которая находится в одной структуре, может быть присвоена другой структуре того же типа при помощи единственного оператора присваивания. Нет необходимости присваивать значения каждого члена в отдельности. Как выполняется присваивание структур, показывает следующая программа:

```
#include <stdio.h>

int main(void)
{
    struct {
        int a;
        int b;
    } x, y;

    x.a = 10;

    y = x; /* присваивание одной структуры другой */

    printf("%d", y.a);

    return 0;
}
```

После присвоения в `y.a` будет храниться значение 10.

Массивы структур

Структуры часто образуют массивы. Чтобы объявить массив структур, вначале необходимо определить структуру (то есть определить агрегатный тип данных), а затем объявить переменную массива этого же типа. Например, чтобы объявить 100-элементный массив структур типа `addr`, который был определен ранее, напишите следующее:

```
struct addr addr_list[100];
```

Это выражение создаст 100 наборов переменных, каждый из которых организован так, как определено в структуре `addr`.

Чтобы получить доступ к определенной структуре, указывайте имя массива с индексом. Например, чтобы вывести ZIP-код из третьей структуры, напишите следующее:

```
printf("%1u", addr_list[2].zip);
```

Как и в других массивах переменных, в массивах структур индексирование начинается с 0.

Для справки: чтобы указать определенную структуру, находящуюся в массиве структур, необходимо указать имя этого массива с определенным индексом. А если нужно указать индекс определенного элемента в структуре, то необходимо указать индекс этого элемента. Таким образом, в результате выполнения следующего выражения первому символу члена `name`, находящегося в третьей структуре из `addr_list`, присваивается значение 'X'.

```
addr_list[2].name[0] = 'X';
```

Пример со списком рассылки

Чтобы показать, как используются структуры и массивы структур, в этом разделе создается простая программа работы со списком рассылки, и в ее массиве структур будут храниться адреса и связанная с ними информация. Эта информация записывается в следующие поля: `name` (имя), `street` (улица), `city` (город), `state` (штат) и `zip` (почтовый код, индекс).

Вся эта информация, как показано ниже, находится в массиве структур типа `addr`:

```
struct addr {
    char name[30];
```



```

char street[40];
char city[20];
char state[3];
unsigned long int zip;
} addr_list[MAX];

```

Обратите внимание, что поле `zip` имеет целый тип `unsigned long`. Правда, чаще можно встретить хранение почтовых кодов, в которых используются строки символов, потому что этот способ подходит для почтовых кодов, в которых вместе с цифрами используются и буквы (как, например, в Канаде и других странах). Однако в нашем примере почтовый индекс хранится в виде целого числа; это делается для того, чтобы показать использование числового элемента в структуре.

Вот `main()` — первая функция, которая нужна программе:

```

int main(void)
{
    char choice;

    init_list(); /* инициализация массива структур */

    for(;;) {
        choice = menu_select();
        switch(choice) {
            case 1: enter();
                    break;
            case 2: delete();
                    break;
            case 3: list();
                    break;
            case 4: exit(0);
        }
    }

    return 0;
}

```

Функция начинает выполнение с инициализации массива структур, а затем реагирует на выбранный пользователем пункт меню.

Функция `init_list()` готовит массив структур к использованию, обнуляя первый байт поля `name` каждой структуры массива. (В программе предполагается, что если поле `name` пустое, то элемент массива не используется.) А вот сама функция `init_list()`:

```

/* Инициализация списка. */
void init_list(void)
{
    register int t;

    for(t=0; t<MAX; ++t) addr_list[t].name[0] = '\0';
}

```

Функция `menu_select()` выводит меню на экран и возвращает то, что выбрал пользователь.

```

/* Получение значения, выбранного в меню. */
int menu_select(void)
{
    char s[80];
    int c;
}

```

```

printf("1. Введите имя\n");
printf("2. Удалите имя\n");
printf("3. Выведите список\n");
printf("4. Выход\n");

do {
    printf("\nВведите номер нужного пункта: ");
    gets(s);
    c = atoi(s);
} while(c<0 || c>4);

return c;
}

```

Функция `enter()` подсказывает пользователю, что именно требуется ввести, и сохраняет введенную информацию в следующей свободной структуре. Если массив заполнен, то выводится сообщение `Список заполнен`. Функция `find_free()` ищет в массиве структур свободный элемент.

```

/* Ввод адреса в список. */
void enter(void)
{
    int slot;
    char s[80];

    slot = find_free();
    if(slot==-1) {
        printf("\nСписок заполнен");
        return;
    }

    printf("Введите имя: ");
    gets(addr_list[slot].name);

    printf("Введите улицу: ");
    gets(addr_list[slot].street);

    printf("Введите город: ");
    gets(addr_list[slot].city);

    printf("Введите штат: ");
    gets(addr_list[slot].state);

    printf("Введите почтовый код: ");
    gets(s);
    addr_list[slot].zip = strtoul(s, '\0', 10);
}

/* Поиск свободной структуры. */
int find_free(void)
{
    register int t;

    for(t=0; addr_list[t].name[0] && t<MAX; ++t) ;

    if(t==MAX) return -1; /* свободных структур нет */
    return t;
}

```

Обратите внимание, что если все элементы массива структур заняты, то `find_free()` возвращает -1. Это удобное число, потому что в массиве нет -1-го элемента.

Функция `delete()` предлагает пользователю указать индекс той записи с адресом, которую требуется удалить. Затем функция обнуляет первый байт поля `name`.

```
/* Удаление адреса. */
void delete(void)
{
    register int slot;
    char s[80];

    printf("Введите № записи: ");
    gets(s);
    slot = atoi(s);
    if(slot >= 0 && slot < MAX)
        addr_list[slot].name[0] = '\0';
}
```

И последняя функция, которая требуется программе, — это `list()`, которая выводит на экран весь список рассылки. Из-за большого разнообразия компьютерных сред язык С не определяет стандартную функцию, которая бы отправляла вывод на принтер. Однако все нужные для этого средства имеются во всех компиляторах С. Возможно, вам самим захочется сделать так, чтобы программа работы со списками могла еще и распечатывать список рассылки.

```
/* Вывод списка на экран */
void list(void)
{
    register int t;

    for(t=0; t<MAX; ++t) {
        if(addr_list[t].name[0]) {
            printf("%s\n", addr_list[t].name);
            printf("%s\n", addr_list[t].street);
            printf("%s\n", addr_list[t].city);
            printf("%s\n", addr_list[t].state);
            printf("%lu\n\n", addr_list[t].zip);
        }
    }
    printf("\n\n");
}
```

Ниже программа обработки списка рассылки приведена полностью. Если у вас остались какие-либо сомнения относительно ее компонентов, введите программу в компьютер и проверьте ее работу, делая в программе изменения и получая соответствующие результаты.

```
/* Простой пример программы обработки списка рассылки, в которой используется массив структур. */
#include <stdio.h>
#include <stdlib.h>

#define MAX 100

struct addr {
    char name[30];
    char street[40];
    char city[20];
    char state[3];
    unsigned long int zip;
};
```

```

} addr_list[MAX];

void init_list(void), enter(void);
void delete(void), list(void);
int menu_select(void), find_free(void);

int main(void)
{
    char choice;

    init_list(); /* инициализация массива структур */

    for(;;) {
        choice = menu_select();
        switch(choice) {
            case 1: enter();
                    break;
            case 2: delete();
                    break;
            case 3: list();
                    break;
            case 4: exit(0);
        }
    }

    return 0;
}

/* Инициализация списка */
void init_list(void)
{
    register int t;

    for(t=0; t<MAX; ++t) addr_list[t].name[0] = '\0';
}

/* Получение значения, выбранного в меню. */
int menu_select(void)
{
    char s[80];
    int c;

    printf("1. Введите имя\n");
    printf("2. Удалите имя\n");
    printf("3. Выведите список\n");
    printf("4. Выход\n");

    do {
        printf("\nВведите номер нужного пункта: ");
        gets(s);
        c = atoi(s);
    } while(c<0 || c>4);
    return c;
}

/* Ввод адреса в список. */
void enter(void)
{
    int slot;

```

```

char s[80];

slot = find_free();
if(slot==-1) {
    printf("\nСписок заполнен");
    return;
}

printf("Введите имя: ");
gets(addr_list[slot].name);

printf("Введите улицу: ");
gets(addr_list[slot].street);

printf("Введите город: ");
gets(addr_list[slot].city);

printf("Введите штат: ");
gets(addr_list[slot].state);

printf("Введите почтовый индекс: ");
gets(s);
addr_list[slot].zip = strtoul(s, '\0', 10);
}

/* Поиск свободной структуры */
int find_free(void)
{
    register int t;

    for(t=0; addr_list[t].name[0] && t<MAX; ++t) ;

    if(t==MAX) return -1; /* свободных структур нет */
    return t;
}

/* Удаление адреса. */
void delete(void)
{
    register int slot;
    char s[80];

    printf("Введите № записи: ");
    gets(s);
    slot = atoi(s);
    if(slot>=0 && slot < MAX)
        addr_list[slot].name[0] = '\0';
}

/* Вывод списка на экран */
void list(void)
{
    register int t;

    for(t=0; t<MAX; ++t) {
        if(addr_list[t].name[0]) {
            printf("%s\n", addr_list[t].name);
            printf("%s\n", addr_list[t].street);
            printf("%s\n", addr_list[t].city);

```

```

        printf("%s\n", addr_list[t].state);
        printf("%lu\n\n", addr_list[t].zip);
    }
    printf("\n\n");
}

```



Передача структур функциям

В этом разделе рассказывается о передаче структур и их членов функциям.

Передача членов структур функциям

При передаче функции члена структуры передается его значение, притом не играет роли то, что значение берется из члена структуры. Проанализируйте, например, следующую структуру:

```

struct fred
{
    char x;
    int y;
    float z;
    char s[10];
} mike;

```

Например, обратите внимание, каким образом каждый член этой структуры передается функции:

```

func(mike.x);      /* передается символьное значение x */
func2(mike.y);     /* передается целое значение y */
func3(mike.z);     /* передается значение с плавающей точкой z */
func4(mike.s);     /* передается адрес строки s */
func(mike.s[2]);   /* передается символьное значение s[2] */

```

В каждом из этих случаев функции передается значение определенного элемента, и здесь не имеет значения то, что этот элемент является частью какой-либо большей совокупности.

Если же нужно передать *адрес* отдельного члена структуры, то перед именем структуры должен находиться оператор `&`. Например, чтобы передать адреса членов структуры `mike`, можно написать следующее:

```

func(&mike.x);     /* передается адрес символа x */
func2(&mike.y);    /* передается адрес целого y */
func3(&mike.z);    /* передается адрес члена z с плавающей точкой */
func4(mike.s);     /* передается адрес строки s */
func(&mike.s[2]);  /* передается адрес символа в s[2] */

```

Обратите внимание, что оператор `&` стоит непосредственно перед именем структуры, а не перед именем отдельного члена. И еще заметьте, что `s` уже обозначает адрес, поэтому `&` не требуется.

Передача целых структур функциям

Когда в качестве аргумента функции используется структура, то для передачи целой структуры используется обычный способ вызова по значению. Это, конечно, означает, что любые изменения в содержимом параметра внутри функции не отразятся на той структуре, которая передана в качестве аргумента.

При использовании структуры в качестве аргумента надо помнить, что тип аргумента должен соответствовать типу параметра. Например, в следующей программе и аргумент `arg`, и параметр `parm` объявляются с одним и тем же типом структуры.

```
#include <stdio.h>

/* Определение типа структуры */
struct struct_type {
    int a, b;
    char ch;
} ;

void f1(struct struct_type parm);

int main(void)
{
    struct struct_type arg;

    arg.a = 1000;

    f1(arg);

    return 0;
}

void f1(struct struct_type parm)
{
    printf("%d", parm.a);
}
```

Как видно из этой программы, при объявлении параметров, являющихся структурами, объявление типа структуры должно быть глобальным, чтобы структурный тип можно было использовать во всей программе. Например, если бы `struct_type` был бы объявлен внутри `main()`, то этот тип не был бы виден в `f1()`.

Как уже говорилось, при передаче структуры тип аргумента должен совпадать с типом параметра. Для аргумента и параметра недостаточно просто быть физически похожими; должны совпадать даже имена их типов. Например, следующая версия предыдущей программы неправильная и компилироваться не будет. Дело в том, что имя типа для аргумента, используемого при вызове функции `f1()`, отличается от имени типа ее параметра.

```
/* Эта программа неправильная и при компиляции будут обнаружены
   ошибки. */
#include <stdio.h>

/* Определение типа структуры */
struct struct_type {
    int a, b;
    char ch;
} ;

/* Определение структуры, похожей на struct_type,
   но с другим именем. */
struct struct_type2 {
    int a, b;
    char ch;
} ;
```

```

void f1(struct struct_type2 parm);

int main(void)
{
    struct struct_type arg;

    arg.a = 1000;

    f1(arg); /* несовпадение типов */

    return 0;
}

void f1(struct struct_type2 parm)
{
    printf("%d", parm.a);
}

```

Указатели на структуры

В языке С указатели на структуры также официально признаны, как и указатели на любой другой вид объектов. Однако указатели на структуры имеют некоторые особенности, о которых и пойдет речь.

Объявление указателя на структуру

Как и другие указатели, указатель на структуру объявляется с помощью звездочки *, которую помещают перед именем переменной структуры. Например, для ранее определенной структуры `addr` следующее выражение объявляет `addr_pointer` указателем на данные этого типа (то есть на данные типа `addr`):

```

struct addr *addr_pointer;

```

Использование указателей на структуры

Указатели на структуры используются главным образом в двух случаях: когда структура передается функции с помощью вызова по ссылке, и когда создаются связанные друг с другом списки и другие структуры с динамическими данными, работающие на основе динамического размещения. В этой главе рассматривается первый случай.

У такого способа, как передача любых (кроме самых простых) структур функциям, имеется один большой недостаток: при выполнении вызова функции, чтобы поместить структуру в стек, необходимы существенные ресурсы. (Вспомните, что аргументы передаются функциям через стек.) Впрочем, для простых структур с несколькими членами эти ресурсы являются не такими уж большими. Но если в структуре имеется большое количество членов или некоторые члены сами являются массивами, то при передаче структур функциям производительность может упасть до недопустимо низкого уровня. Как же решить эту проблему? Надо передавать не саму структуру, а указатель на нее.

Когда функции передается указатель на структуру, то в стек попадает только адрес структуры. В результате вызовы функции выполняются очень быстро. В некоторых случаях этот способ имеет еще и второе преимущество: передача указателя позволяет функции модифицировать содержимое структуры, используемой в качестве аргумента.

Чтобы получить адрес переменной-структуры, необходимо перед ее именем поместить оператор `&`. Например, в следующем фрагменте кода


```
struct bal {
    float balance;
    char name[80];
};
```

```
struct bal *p; /* объявление указателя на структуру */
```

адрес структуры person можно присвоить указателю p:

```
p = &person;
```

Чтобы с помощью указателя на структуру получить доступ к ее членам, необходимо использовать оператор стрелка ->. Вот, например, как можно сослаться на поле balance:

```
p->balance
```

Оператор ->, который обычно называют *оператором стрелки*, состоит из знака “минус”, за которым следует знак “больше”. Стрелка применяется вместо оператора точки тогда, когда для доступа к члену структуры используется указатель на структуру.

Чтобы увидеть, как можно использовать указатель на структуру, проанализируйте следующую простую программу, которая имитирует таймер, выводящий значения часов, минут и секунд:

```
/* Программа-имитатор таймера. */
#include <stdio.h>

#define DELAY 128000

struct my_time {
    int hours;
    int minutes;
    int seconds;
} ;

void display(struct my_time *t);
void update(struct my_time *t);
void delay(void);

int main(void)
{
    struct my_time Начало systime;
    10:50
    systime.hours = 0;
    systime.minutes = 0;
    systime.seconds = 0;
    10:50
    for(;;) {
        update(&systime);
        display(&systime);
    }

    return 0;
}

void update(struct my_time *t)
{
    t->seconds++;
    if(t->seconds==60) {
        t->seconds = 0;
        t->minutes++;
    }
}
```

```

    }

    if(t->minutes==60) {
        t->minutes = 0;
        t->hours++;
    }

    if(t->hours==24) t->hours = 0;
    delay();
}

void display(struct my_time *t)
{
    printf("%02d:", t->hours);
    printf("%02d:", t->minutes);
    printf("%02d\n", t->seconds);
}

void delay(void)
{
    long int t;

    /* если надо, можете изменить константу DELAY (задержка) */
    for(t=1; t<DELAY; ++t) ;
}

```

Эту программу можно настраивать, меняя определение DELAY.

В этой программе объявлена глобальная структура `my_time`, но при этом не объявлены никакие другие переменные программы. Внутри же `main()` объявлена структура `systemtime` и она инициализируется значением 00:00:00. Это значит, что `systemtime` непосредственно видна только в функции `main()`.

Функциям `update()` (которая изменяет значения времени) и `display()` (которая выводит эти значения) передается адрес структуры `systemtime`. Аргументы в обеих функциях объявляются как указатель на структуру `my_time`.

Внутри `update()` и `display()` доступ к каждому члену `systemtime` осуществляется с помощью указателя. Так как функция `update()` принимает указатель на структуру `systemtime`, то она в состоянии обновлять значение этой структуры. Например, необходимо “в полночь”, когда значение переменной, в которой хранится количество часов, станет равным 24, сбросить отсчет и снова сделать значение этой переменной равным 0. Для этого в `update()` имеется следующая строка:

```

if(t->hours==24) t->hours = 0;

```

Таким образом, компилятору дается указание взять адрес `t` (этот адрес указывает на переменную `systemtime` из `main()`) и сбросить значение `hours` в нуль.

Помните, что оператор точка используется для доступа к элементам структуры при работе с самой структурой. А когда используется указатель на структуру, то надо применять оператор стрелка.



Массивы и структуры внутри структур

Членом структуры может быть или простая переменная, например, типа `int` или `double`, или составной (не скалярный) тип. В языке C составными типами являются массивы и структуры. Один составной тип вы уже видели — это символьные массивы, которые использовались в `addr`.

Члены структуры, которые являются массивами, можно считать такими же членами структуры, как и те, что нам известны из предыдущих примеров. Например, проанализируйте следующую структуру:

```
struct x {  
    int a[10][10]; /* массив 10 x 10 из целых значений */  
    float b;  
} y;
```

Целый элемент с индексами 3, 7 из массива *a*, находящегося в структуре *y*, обозначается таким образом:

```
y.a[3][7]
```

Когда структура является членом другой структуры, то она называется *вложенной*. Например, в следующем примере структура *address* вложена в *emp*:

```
struct emp {  
    struct addr address; /* вложенная структура */  
    float wage;  
} worker;
```

Здесь структура была определена как имеющая два члена. Первым является структура типа *addr*, в которой находится адрес работника. Второй член — это *wage*, где находятся данные по его зарплате. В следующем фрагменте кода элементу *zip* из *address* присваивается значение 93456.

```
worker.address.zip = 93456;
```

Как вы видите, в каждой структуре любой член обозначают с помощью тех структур, в которые он вложен — начиная от самых общих и заканчивая той, непосредственно в которой он находится. В соответствии со стандартом C89 структуры могут быть вложенными вплоть до 15-го уровня. А стандарт C99 допускает уровень вложенности до 63-го включительно.



Объединения

Объединение — это место в памяти, которое используется для хранения переменных разных типов. Объединение дает возможность интерпретировать один и тот же набор битов не менее, чем двумя разными способами. Объявление объединения (начинается с ключевого слова *union*) похоже на объявление структуры и в общем виде выглядит так:

```
union тег {  
    тип имя-члена;  
    тип имя-члена;  
    тип имя-члена;  
    .  
    .  
    .  
} переменные-этого-объединения;
```

Например:

```
union u_type {  
    int i;  
    char ch;  
};
```

Это объявление не создает никаких переменных. Чтобы объявить переменную, ее имя можно поместить в конце объявления или написать отдельный оператор объявления. Чтобы с помощью только что написанного кода объявить переменную-объединение, которая называется `cnvt` и имеет тип `u_type`, можно написать следующий оператор:

```
union u_type cnvt;
```

В `cnvt` одну и ту же область памяти занимают целая переменная `i` и символьная переменная `ch`. Конечно, `i` занимает 2 байта (при условии, что целые значения занимают по 2 байта), а `ch` — только 1. На рис. 7.2 показано, каким образом `i` и `ch` пользуются одним и тем же адресом. В любом месте программы хранящиеся в `cnvt` данные можно обрабатывать как целые или символьные.

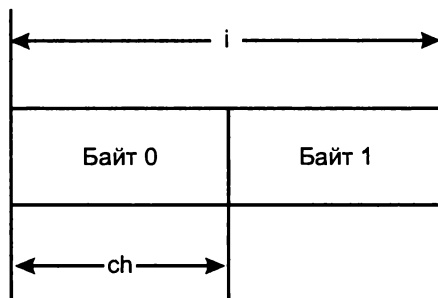


Рис. 7.2. Как `i`, так и `ch`, хранятся в объединении `cnvt` (подразумевается, что целые значения занимают по 2 байта)

Когда переменная объявляется с ключевым словом `union`, компилятор автоматически выделяет столько памяти, чтобы в ней поместился самый большой член нового объединения. Например, при условии, что целые значения занимают по 2 байта, для размещения `i` в `cnvt` необходимо, чтобы длина этого объединения составляла 2 байта, даже если для `ch` требуется только 1 байт.

Для получения доступа к члену объединения используйте тот же синтаксис, что и для структур: операторы точки и стрелки. При работе непосредственно с объединением следует пользоваться точкой. А при получении доступа к объединению с помощью указателя нужен оператор стрелка. Например, чтобы присвоить целое значение 10 элементу `i` из `cnvt`, напишите

```
cnvt.i = 10;
```

В следующем примере функции `func1` передается указатель на `cnvt`:

```
void func1(union u_type *un)
{
    un->i = 10; /* присвоение cnvt значения 10 с помощью указателя */
}
```

Объединения часто используются тогда, когда нужно выполнить специфическое преобразование типов, потому что хранящиеся в объединениях данные можно обозначать совершенно разными способами. Например, используя объединения, можно манипулировать байтами, составляющими значение типа `double`, и делать так, чтобы менять его точность или выполнять какое-либо необычное округление.

Чтобы получить представление о полезности объединений в случаях, когда нужны нестандартные преобразования типа, подумайте над проблемой записи целых значений типа `short` в файл, который находится на диске.

В стандартной библиотеке языка С не определено никакой функции, специально предназначенной для выполнения этой записи.

Хотя данные любого типа можно записывать в файл, пользуясь функцией `fwrite()`, но было бы нерационально применять этот способ для такой простой операции, как запись на диск целых значений типа `short`, так как получится чрезмерный перерасход ресурсов. А вот, используя объединение, можно легко создать функцию `putw()`, которая по одному байту будет записывать в файл двоичное представление целого значения типа `short`. (В этом примере предполагается, что такие значения имеют длину 2 байта каждое.) Чтобы увидеть, как это делается, вначале создадим объединение, состоящее из целой переменной типа `short` и из массива 2-байтовых символов:

```
union pw {
    short int i;
    char ch[2];
};
```

Теперь с помощью `pw` можно написать вариант `putw()`, приведенный в следующей программе.

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

union pw {
    short int i;
    char ch[2];
};

int putw(short int num, FILE *fp);

int main(void)
{
    FILE *fp;

    fp = fopen("test.tmp", "wb+");
    if(fp == NULL) {
        printf("Файл не открыт.\n");
        exit(1);
    }

    putw(1025, fp); /* записать значение 1025 */
    fclose(fp);

    return 0;
}

int putw(short int num, FILE *fp)
{
    union pw word;

    word.i = num;

    putc(word.ch[0], fp); /* записать первую половину */
    return putc(word.ch[1], fp); /* записать вторую половину */
}
```

Хотя функция `putw()` и вызывается с целым аргументом типа `short`, ей для выполнения побайтовой записи в файл на диске все равно приходится использовать стандартную функцию `putc()`.



Битовые поля

В отличие от некоторых других компьютерных языков, в языке С имеется встроенная поддержка *битовых полей*¹, которая дает возможность получать доступ к единичному биту. Битовые поля могут быть полезны по разным причинам, а именно:

- Если память ограничена, то в одном байте можно хранить несколько булевых переменных (принимающих значения ИСТИНА и ЛОЖЬ);
- Некоторые устройства передают информацию о состоянии, закодированную в байте в одном или нескольких битах;
- Для некоторых процедур шифрования требуется доступ к отдельным битам внутри байта.

Хотя для решения этих задач можно успешно применять побитовые операции, битовые поля могут придать вашему коду больше упорядоченности (и, возможно, с их помощью удастся достичь большей эффективности).

Битовое поле может быть членом структуры или объединения. Оно определяет длину поля в битах. Общий вид определения битового поля такой:

тип имя : длина;

Здесь *тип* означает тип битового поля, а *длина* — количество бит, которые занимает это поле. Тип битового поля может быть `int`, `signed` или `unsigned`. (Кроме того, в соответствии со стандартом С99, у битового поля еще может быть тип `_Bool`.)

Битовые поля часто используются при анализе данных, поступающих в программу с аппаратуры. Например, в результате опроса состояния адаптера последовательной связи может возвращаться байт состояния, организованный следующим образом:

Бит Что означает, если установлен

- | | |
|---|--|
| 0 | Изменение в линии сигнала разрешения на передачу (change in clear-to-send line) |
| 1 | Изменение состояния готовности устройства сопряжения (change in data-set-ready) |
| 2 | Обнаружена концевая запись (trailing edge detected) |
| 3 | Изменение в приемной линии (change in receive line) |
| 4 | Разрешение на передачу. Сигналом CTS (clear-to-send) модем разрешает подключенному терминалу передавать данные |
| 5 | Модем готов (data-set-ready) |
| 6 | Телефонный вызов (telephone ringing) |
| 7 | Сигнал принят (received signal) |

Информацию в байте состояния можно представить с помощью следующего битового поля:

```
struct status_type {
    unsigned delta_cts: 1;
    unsigned delta_dsr: 1;
    unsigned tr_edge: 1;
    unsigned delta_rec: 1;
    unsigned cts: 1;
    unsigned dsr: 1;
    unsigned ring: 1;
    unsigned rec_line: 1;
} status;
```

¹ Называются также *полями битов*. — Прим. ред.

Для того чтобы программа могла определить, когда можно отправлять или принимать данные, можно использовать такие операторы:

```
status = get_port_status();
if(status.cts) printf("Разрешение на передачу");
if(status.dsr) printf("Данные готовы");
```

Для присвоения битовому полю значения используйте тот же способ, что и для элемента, находящегося в структуре любого другого типа. Вот, например, фрагмент кода, выполняющий сброс поля ring:

```
status.ring = 0;
```

Как видно из этого примера, каждое битовое поле доступно с помощью оператора точка. Однако если структура передана с помощью указателя, то следует использовать оператор стрелка ->.

Нет необходимости давать имя каждому битовому полю. Таким образом можно легко получать доступ к нужному биту, обходя неиспользуемые. Например, если вас интересуют только биты cts и dsr, то структуру status_type можно объявить таким образом:

```
struct status_type {
    unsigned : 4;
    unsigned cts: 1;
    unsigned dsr: 1;
} status;
```

Кроме того, обратите внимание, что если биты, расположенные после dsr, не используются, то определять их не надо.

В структурах можно сочетать обычные члены с битовыми полями. Например, в структуре

```
struct emp {
    struct addr address;
    float pay;
    unsigned lay_off: 1; /* временно уволенный или работающий */
    unsigned hourly: 1; /* почасовая оплата или оклад */
    unsigned deductions: 3; /* налоговые (IRS) удержания */
};
```

определены данные о работнике, для которых выделяется только один байт, содержащий информацию трех видов: статус работника, на окладе ли он, а также количество удержаний из его зарплаты. Без битового поля эта информация занимала бы 3 байта.

Использование битовых полей имеет определенные ограничения. Нельзя получить адрес битового поля. Нет массивов битовых данных. При переносе кода на другую машину неизвестно, будут ли поля обрабатываться справа налево или слева направо; это значит, что выполнение любого кода, в котором используются битовые поля, в определенной степени может зависеть от машины, на которой он выполняется. Другие ограничения будут зависеть от конкретных реализаций.



Перечисления

Перечисление — это набор именованных целых констант. Перечисления довольно часто встречаются в повседневной жизни. Вот, например, перечисление, в котором приведены названия монет, используемых в Соединенных Штатах:

penny (пенни, монета в один цент), nickel (никель, монета в пять центов), dime (монета в 10 центов), quarter (25 центов, четверть доллара), half-dollar (полдоллара), dollar (доллар)

Перечисления определяются во многом так же, как и структуры; началом объявления перечислимого типа¹ служит ключевое слово `enum`. Перечисление в общем виде выглядит так:

```
enum тег { список перечисления } список переменных;
```

Здесь `тег` и список переменных не являются обязательными. (Но хотя бы что-то одно из них должно присутствовать.) Следующий фрагмент кода определяет перечисление с именем `coin` (монета):

```
enum coin { penny, nickel, dime, quarter, half_dollar, dollar };
```

`Тег` перечисления можно использовать для объявления переменных данного перечислимого типа. Вот код, в котором `money` (деньги) объявляется в качестве переменной типа `coin`:

```
enum coin money;
```

С учетом этих объявлений совершенно верными являются следующие операторы:

```
money = dime;
if(money == quarter) printf("Денег всего четверть доллара.\n");
```

Главное, что нужно знать для понимания перечислений — каждый их элемент² представляет целое число. В таком виде элементы перечислений можно применять везде, где используются целые числа. Каждому элементу дается значение, на единицу большее, чем у его предшественника. Первый элемент перечисления имеет значение 0. Поэтому, при выполнении кода

```
printf("%d %d", penny, dime);
```

на экран будет выведено 0 2.

Однако для одного или более элементов можно указать значение, используемое как инициализатор. Для этого после перечислителя надо поставить знак равенства, а затем — целое значение. Перечислителям, которые идут после инициализатора, присваиваются значения, большие предшествующего. Например, следующий код присваивает `quarter` значение 100:

```
enum coin { penny, nickel, dime, quarter=100, half_dollar, dollar};
```

И вот какие значения появились у этих элементов:

penny	0	
nickel	1	
dime	2	
quarter	100	
half_dollar	101	
dollar	102	

Относительно перечислений есть одно распространенное, но ошибочное мнение. Оно состоит в том, что их элементы можно непосредственно вводить и выводить. Это не так. Например, следующий фрагмент кода не будет выполняться так, как того ожидают многие неопытные программисты:

```
/* этот код работать не будет */
money = dollar;
printf("%s", money);
```

¹ Иногда используется термин *перечисляемый тип*. — Прим. ред.

² Элементы списка перечисления называются также *перечислителями* и *идентификаторами*. — Прим. ред.

Здесь `dollar` — это имя для значения целого типа; это не строка. Таким образом, попытка вывести `money` в виде строки по существу обречена. По той же причине для достижения нужных результатов не годится и такой код:

```
/* этот код неправильный */
strcpy(money, "dime");
```

То есть строка, содержащая имя элемента, автоматически в этот перечислитель не превратится.

На самом же деле создавать код для ввода и вывода элементов перечислений — это довольно-таки скучное занятие (но его можно избежать лишь тогда, когда будет достаточно именно целых значений этих перечислителей). Например, чтобы вывести название монеты, вид которой находится в `money`, потребуется следующий код:

```
switch(money) {
    case penny: printf("пенни");
                break;
    case nickel: printf("никель");
                break;
    case dime: printf("монета в 10 центов");
                break;
    case quarter: printf("четверть доллара");
                break;
    case half_dollar: printf("полдоллара");
                break;
    case dollar: printf("доллар");
}
}
```

Иногда можно объявить строчный массив и использовать значение перечисления как индекс при переводе этого значения в соответствующую строку. Например, следующий код также выводит нужную строку:

```
char name[][12]={
    "пенни",
    " никель",
    " монета в 10 центов",
    "четверть доллара",
    "полдоллара",
    "доллар"
};
printf("%s", name[money]);
```

Конечно, он будет работать только тогда, когда не инициализирован ни один из элементов перечисления, так как строчный массив должен иметь индекс, который начинается с 0 и возрастает каждый раз на 1.

Так как при операциях ввода/вывода необходимо специально заботиться о преобразовании перечислений в их строчный эквивалент, который можно легко прочитать, то перечисления полезнее всего именно в тех процедурах, где такие преобразования не нужны. Например, перечисления часто применяются, чтобы определить таблицы соответствия символов в компиляторах.



Важное различие между C и C++

Что касается имен типов структур, объединений и перечислений, то между языками C и C++ имеется важное различие. Чтобы понять эту разницу, проанализируйте следующее объявление структуры:

```
struct MyStruct {
    int a;
    int b;
};
```

В языке С имя `MyStruct` называется *тегом*. Чтобы объявить объект типа `MyStruct`, необходимо использовать выражение, аналогичное следующему:

```
struct MyStruct obj;
```

Как можно заметить, перед именем `MyStruct` находится ключевое слово `struct`. Однако в С++ для этой операции достаточно использовать объявление покороче:

```
MyStruct obj; /* Нормально для С++, неправильно для С */
```

В С++ не требуется ключевое слово `struct`. В С++, как только структура объявлена, можно объявлять переменные ее типа, используя только ее тег и не ставя перед ним ключевого слова `struct`. Дело здесь в том, что в С имя структуры не определяет полное имя типа. Вот почему в С это имя называется не полным именем, а тегом. Однако в С++ имя структуры является полным именем типа, и оно может использоваться для определения переменных. Не надо, впрочем, забывать, что до сих пор можно на вполне законных основаниях в программе на языке С++ использовать и объявление в стиле С.

Все сказанное выше можно обобщить для объединений и перечислений. Таким образом, в С при объявлении объектов непосредственно перед тегом имени должно находиться одно из ключевых слов: `struct`, `union` или `enum` (в зависимости от конкретного случая). А в С++ ключевое слово не требуется.

Так как для С++ подходят объявления в стиле С, то во время переноса программ из С в С++ по этому поводу беспокоиться нечего. Но при переносе из С++ в С соответствующие изменения сделать придется.

■ Использование `sizeof` для обеспечения переносимости

Вы имели возможность убедиться, что структуры и объединения можно использовать для создания переменных разных размеров, а также в том, что настоящий размер этих переменных в разных машинах может быть разным. Оператор `sizeof` подсчитывает размер любой переменной или любого типа и может быть полезен, если в программах требуется свести к минимуму машинно-зависимый код. Этот оператор особенно полезен там, где приходится иметь дело со структурами или объединениями.

Перед тем как переходить к дальнейшему изложению, предположим, что определенные типы данных имеют следующие размеры:

Тип	Размер в байтах
<code>char</code>	1
<code>int</code>	4
<code>double</code>	8

Поэтому при выполнении следующего кода на экран будут выведены числа 1, 4 и 8:

```
char ch;
int i;
double f;

printf("%d", sizeof(ch));
```

```
printf("%d", sizeof(i));
printf("%d", sizeof(f));
```

Размер структуры равен сумме размеров ее членов или, возможно, даже *больше* этой суммы. Рассмотрим пример:

```
struct s {
    char ch;
    int i;
    double f;
} s_var;
```

Здесь `sizeof(s_var)` равняется как минимум 13 ($=8+4+1$). Однако размер `s_var` может быть и больше, потому что компилятору иногда необходимо специально увеличить размер структуры, выровнять некоторые ее члены на границу слова или параграфа. (Параграф занимает 16 байтов.) Так как размер структуры может быть больше, чем сумма размеров ее членов, то всегда, когда нужно знать размер структуры, следует использовать `sizeof`. Например, если требуется динамически выделять память для объекта типа `s`, необходимо использовать последовательность операторов, аналогичную той, что показана здесь (а не вставлять вручную значения длины его членов):

```
struct s *p;
p = malloc(sizeof(struct s));
```

Так как `sizeof` — это оператор времени компиляции, то вся информация, необходимая для вычисления размера любой переменной, становится известной как раз во время компиляции. Это особенно важно для объединений, потому что размер каждого из них всегда равен размеру наибольшего члена. Например, проанализируйте следующее объединение:

```
union u {
    char ch;
    int i;
    double f;
} u_var;
```

Для него `sizeof(u_var)` равняется 8. Впрочем, во время выполнения не имеет значения, какой размер на самом деле имеет `u_var`. Важен размер его наибольшего члена, так как любое объединение должно быть такого же размера, как и его самый большой элемент.



Средство typedef

Новые имена типов данных можно определять, используя ключевое слово `typedef`. На самом деле таким способом новый тип данных не создается, а всего лишь определяется новое имя для уже существующего типа. Этот процесс может помочь сделать машинно-зависимые программы более переносимыми. Если вы для каждого машинно-зависимого типа данных, используемого в вашей программе, определяете данное вами имя, то при компиляции для новой среды придется менять только операторы `typedef`. Такие выражения могут помочь в самодокументировании кода, позволяя давать понятные имена стандартным типам данных. Общий вид декларации `typedef` (оператора `typedef`) такой:

```
typedef тип новое_имя;
```

где *тип* — это любой тип данных языка C, а *новое_имя* — новое имя этого типа. Новое имя является дополнением к уже существующему, а не его заменой.

Например, для `float` можно создать новое имя с помощью

```
■ typedef float balance;
```

Это выражение дает компилятору указание считать `balance` еще одним именем `float`. Затем, используя `balance`, можно создать переменную типа `float`:

```
■ balance over_due;
```

Теперь имеется переменная с плавающей точкой `over_due` типа `balance`, а `balance` является еще одним именем типа `float`.

Теперь, когда имя `balance` определено, его можно использовать и в другом операторе `typedef`. Например, выражение

```
■ typedef balance overdraft;
```

дает компилятору указание признавать `overdraft` в качестве еще одного имени `balance`, которое в свою очередь является еще одним именем `float`.

Использование операторов `typedef` может облегчить чтение кода и его перенос на новую машину. Однако новый физический тип данных таким способом вы не создадите.

Полный
справочник по



Глава 8

Ввод/вывод на консоль

В языке С не определено никаких ключевых слов, с помощью которых можно выполнять ввод/вывод. Вместо них используются библиотечные функции. Система ввода/вывода языка С — это элегантная конструкция, которая обеспечивает гибкий и в то же время сложенный механизм передачи данных от одного устройства к другому. Впрочем, эта система достаточно большая и состоит из нескольких различных функций. Заголовочным файлом для функций ввода/вывода является `<stdio.h>`.

Имеются как консольные, так и файловые¹ функции ввода/вывода. С практической точки зрения консольный и файловый ввод/вывод отличаются друг от друга очень мало. Однако теоретически они находятся в двух очень “разных мирах”. В этой главе подробно рассказывается о функциях ввода/вывода на консоль. В следующей же главе представлена система файлового ввода/вывода, а также говорится, что имеется общего между этими двумя системами.

За одним исключением, в этой главе рассказывается только о функциях ввода/вывода на консоль, которые определяются стандартом языка С. В стандарте языка С не определены никакие функции, предназначенные для выполнения различных операций управления экраном (например, позиционирования курсора) или вывода на него графики. И не определены потому, что эти операции на разных машинах очень сильно отличаются. Кроме того, в стандартном С не определены никакие функции, которые выполняют операции вывода в обычном или диалоговом окне, создаваемом в среде Windows. Функции ввода/вывода на консоль выполняют всего лишь телетайпный вывод. Однако в библиотеках большинства компиляторов имеются функции графики и управления экраном, предназначенные для той среды, в которой как раз и должны выполняться программы. И, конечно же, на языке С можно писать Windows-программы. Просто в С не определены функции, которые выполняли бы эти задачи напрямую.

В этой главе консольными функциями ввода/вывода называются те, которые выполняют ввод с клавиатуры и вывод на экран. В действительности же эти функции работают со стандартным потоком ввода и *стандартным потоком вывода*². Более того, *стандартный ввод*³ и *стандартный вывод*⁴ могут быть перенаправлены на другие устройства. Таким образом, “консольные функции” не обязательно должны работать только с консолью. Перенаправление ввода/вывода описано в главе 9. В данной же главе предполагается, что ни стандартный ввод, ни стандартный вывод на другие устройства не перенаправляются.

На заметку

В языке С++ ввод/вывод выполняют не только функции, но имеются еще и операторы (знаки операций. — Прим. ред.) ввода/вывода. Впрочем, в языке С эти операторы ввода/вывода не поддерживаются.

¹ Ввод/вывод в файл и файловый ввод/вывод — синонимы. — Прим. ред.

² Иногда говорят, что ввод происходит со стандартного устройства ввода, а вывод — на стандартное устройство вывода. — Прим. ред.

³ Стандартный ввод — логический файл для ввода данных, связываемый с физическим файлом или стандартным выводом другой программы при запуске. По умолчанию стандартный ввод в пакетном режиме связывается со входным потоком, а в диалоговом режиме — с терминалом. — Прим. ред.

⁴ Стандартный вывод — логический файл вывода данных, связываемый с физическим файлом или стандартным вводом другой программы при запуске. По умолчанию стандартный вывод в пакетном режиме связывается с выходным потоком, а в диалоговом режиме — с терминалом. — Прим. ред.

■ Чтение и запись символов

Самыми простыми из консольных функций ввода/вывода являются `getchar()`, которая читает символ с клавиатуры, и `putchar()`, которая отображает символ на экране. Первая из этих функций ожидает, пока не будет нажата клавиша, а затем возвращает значение этой клавиши. Кроме того, при нажатии клавиши на клавиатуре на экране дисплея автоматически отображается соответствующий символ. Вторая же функция, `putchar()`, отображает символ на экране в текущей позиции курсора. Вот прототипы функций `getchar()` и `putchar()`:

```
int getchar(void);
int putchar(int c);
```

Как видно из прототипа, считается, что функция `getchar()` возвращает целый результат. Однако возвращаемое значение можно присвоить переменной типа `char`, что обычно и делается, так как символ содержится в младшем байте. (Старший байт при этом обычно обнулен.) В случае ошибки `getchar()` возвращает EOF. (Макрос EOF определяется в `<stdio.h>` и часто равен -1.)

Что же касается `putchar()`, то несмотря на то, что эта функция объявлена как принимающая целый параметр, она обычно вызывается с символьным аргументом. На самом деле из ее аргумента на экран выводится только младший байт. Функция `putchar()` возвращает записанный символ или, в случае ошибки, EOF.

В следующей программе продемонстрировано применение `getchar()` и `putchar()`. В этой программе с клавиатуры вводятся символы, а затем они отображаются на другом регистре. То есть символы, вводимые на верхнем регистре, выводятся на нижнем, а вводимые на нижнем — выводятся на верхнем. Чтобы остановить программу, введите точку.

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

int main(void)
{
    char ch;

    printf("Введите какой-нибудь текст (для завершения работы введите точку).\n");
    do {
        ch = getchar();

        if(islower(ch)) ch = toupper(ch);
        else ch = tolower(ch);

        putchar(ch);
    } while (ch != '.');

    return 0;
}
```

(Эта программа не работает, правда, с кириллическими символами. — *Прим. перев.*)

Трудности использования `getchar()`

Использование `getchar()` может быть связано с определенными трудностями. Во многих библиотеках компиляторов эта функция реализуется таким образом, что она заполняет буфер ввода до тех пор, пока не будет нажата клавиша <ENTER>. Это назы-

вается *построчно буферизованным вводом*. Чтобы функция `getchar()` возвратила какой-либо символ, необходимо нажать клавишу `<ENTER>`. Кроме того, эта функция при каждом ее вызове вводит только по одному символу. Поэтому сохранение в буфере целой строки может привести к тому, что в очереди на ввод останутся ждать один или несколько символов, а в интерактивной среде это раздражает достаточно сильно. Хотя `getchar()` и можно использовать в качестве интерактивной функции, но это делается редко. Так что если предшествующая программа ведет себя не так, как ожидалось, то вы теперь знаете, в чем тут дело.

Альтернативы `getchar()`

Так как `getchar()`, имеющаяся в библиотеке компилятора, может оказаться неподходящей в интерактивной среде, то для чтения символов с клавиатуры может потребоваться другая функция. В стандарте языка C не определяется никаких функций, которые гарантировали бы интерактивный ввод, но их определения имеются буквально в библиотеках всех компиляторов C. И пусть в стандарте C эти функции не определены, но известны они всем! А известны они благодаря функции `getchar()`, которая для большинства программистов явно не подходит.

У двух из самых распространенных альтернативных функций `getch()` и `getche()` имеются следующие прототипы:

```
int getch(void);
int getche(void);
```

В библиотеках большинства компиляторов прототипы таких функций находятся в заголовочном файле `<conio.h>`. В библиотеках некоторых компиляторов имена этих функций начинаются со знака подчеркивания (`_`). Например, в Visual C++ компании Microsoft они называются `_getch()` и `_getche()`.

Функция `getch()` ожидает нажатия клавиши, после которого она немедленно возвращает значение. Причем, символ, введенный с клавиатуры, на экране не отображается. Имеется еще и функция `getche()`, которая хоть и такая же, как `getch()`, но символ на экране отображает. И если в интерактивной программе необходимо прочесть символ с клавиатуры, то часто вместо `getchar()` применяется `getche()` или `getch()`. Вот, например, предыдущая программа, в которой `getchar()` заменена функцией `getch()`:

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

int main(void)
{
    char ch;

    printf("Введите какой-нибудь текст (для завершения работы введите точку) .\n");
    do {
        ch = getch();

        if(islower(ch)) ch = toupper(ch);
        else ch = tolower(ch);

        putchar(ch);
    } while (ch != '.');

    return 0;
}
```


Когда выполняется эта версия программы, при каждом нажатии клавиши соответствующий символ сразу передается программе и выводится на другом регистре. А ввод в строках не буферизируется. И хотя в кодах в этой книге функции `getch()` и `getche()` больше не встречаются, но они вполне могут пригодиться в тех программах, которые напишете вы.

На заметку

Тогда, когда писались эти слова, при использовании компилятора Visual C++ компании Microsoft функции `_getch()` и `_getche()` были несовместимы с функциями ввода стандартного C, например, с функциями `scanf()` или `gets()`. Поэтому вам придется вместо стандартных функций использовать такие их специальные версии, как `cscanf()` или `cgets()`. Чтобы получить более подробную информацию, следует изучить документацию по Visual C++.



Чтение и запись строк

Среди функций ввода/вывода на консоль есть и более сложные, но и более мощные: это функции `gets()` и `puts()`, которые позволяют считывать и отображать строки символов.

Функция `gets()` читает строку символов, введенную с клавиатуры, и записывает ее в память по адресу, на который указывает ее аргумент. Символы можно вводить с клавиатуры до тех пор, пока не будет введен символ возврата каретки. Он не станет частью строки, а вместо него в ее конец будет помещен символ конца строки (`'\0'`), после чего произойдет возврат из функции `gets()`. На самом деле вернуть символ возврата каретки с помощью этой функции нельзя (а с помощью `getchar()` — как раз можно). Перед тем как нажимать `<ENTER>`, можно исправлять неправильно введенные символы, пользуясь для этого клавишей возврата каретки на одну позицию (клавишей `backspace`). Вот прототип для `gets()`:

```
char *gets(char *cmp);
```

Здесь `cmp` — это указатель на массив символов, в который записываются символы, вводимые пользователем. `gets()` также возвращает `cmp`. Следующая программа читает строку в массив `str` и выводит ее длину:

```
#include <stdio.h>
#include <string.h>
int main(void)
{
    char str[80];

    gets(str);
    printf("Длина в символах равна %d", strlen(str));

    return 0;
}
```

Необходимо очень осторожно использовать `gets()`, потому что эта функция не проверяет границы массива, в который записываются введенные символы. Таким образом, может случиться, что пользователь введет больше символов, чем помещается в этом массиве. Хотя функция `gets()` прекрасно подходит для программ-примеров и простых утилит, предназначенных только для вас, но в профессиональных программах ею лучше не пользоваться. Ее альтернативой, позволяющей предотвратить переполнение массива, будет функция `fgets()`, которая описана в следующей главе.

Функция `puts()` отображает на экране свой строковый аргумент, после чего курсор переходит на новую строку. Вот прототип этой функции:

```
int puts(const char *cmp);
```

puts() признает те же самые *управляющие последовательности*¹, что и printf(), например, \t в качестве символа табуляции. Вызов функции puts() требует намного меньше ресурсов, чем вызов printf(). Это объясняется тем, что puts() может только выводить строку символов, но не может выводить числа или делать преобразования формата. В результате эта функция занимает меньше места и выполняется быстрее, чем printf(). Поэтому тогда, когда не нужны преобразования формата, часто используется функция puts().

Функция puts() в случае успешного завершения возвращает неотрицательное значение, а в случае ошибки — EOF. Однако при записи на консоль обычно предполагают, что ошибки не будет, поэтому значение, возвращаемое puts(), проверяется редко. Следующий оператор выводит фразу Привет:

```
puts("Привет");
```

В таблице 8.1 перечислены основные функции консольного ввода/вывода.

Таблица 8.1. Основные функции ввода/вывода

Функция	Ее действия
getchar()	Читает символ с клавиатуры; обычно ожидает возврат каретки
getche()	Читает символ, при этом он отображается на экране; не ожидает возврата каретки; в стандарте C не определена, но распространена достаточно широко
getch()	Читает символ, но не отображает его на экране; не ожидает возврата каретки; в стандарте C не определена, но распространена достаточно широко
putchar()	Отображает символ на экране
gets()	Читает строку с клавиатуры
puts()	Отображает строку на экране

В следующей программе — простом компьютеризованном словаре — показано применение нескольких основных функций консольного ввода/вывода. Эта программа предлагает пользователю ввести слово, а затем проверяет, совпадает ли оно с каким-либо из тех слов, что находятся в ее базе данных. Если оно там есть, то программа выводит значение слова. Обратите особое внимание на использование косвенной адресации в этой программе. Чтобы легче было понять программу, прежде всего вспомните, что массив dic — это массив указателей на строки. Обратите внимание, что список должен завершаться двумя нулями.

```
/* Простой словарь. */
#include <stdio.h>
#include <string.h>
#include <ctype.h>

/* список слов и их значений */
char *dic[][40] = {
    "атлас", "Том географических и/или топографических карт.",
    "автомобиль", "Моторизованное средство передвижения.",
    "телефон", "Средство связи.",
    "самолет", "Летающая машина.",
    "", "" /* нули, завершающие список */
};
```

¹ Называются также *ESC-последовательностями*; в C/C++ — это комбинация символов, обычно используемая для задания неотображаемых символов и символов, имеющих специальное значение. Представление управляющих последовательностей начинается с обратной косой черты. — Прим. ред.

```

int main(void)
{
    char word[80], ch;
    char **p;

    do {
        puts("\nВведите слово: ");
        scanf("%s", word);

        p = (char **)dic;

        /* поиск слова в словаре и вывод его значения */
        do {
            if(!strcmp(*p, word)) {
                puts("Значение:");
                puts(*(p+1));
                break;
            }
            if(!strcmp(*p, word)) break;
            p = p + 2; /* продвижение по списку */
        } while(*p);
        if(!*p) puts("Слово в словаре отсутствует.");
        printf("Будете еще вводить? (y/n): ");
        scanf(" %c%c", &ch);
    } while(toupper(ch) != 'N');

    return 0;
}

```



Форматный ввод/вывод на консоль

Функции `printf()` и `scanf()` выполняют форматный ввод и вывод, то есть они могут читать и писать данные в разных форматах. Данные на консоль выводит `printf()`. А ее “дополнение”, функция `scanf()`, наоборот — считывает данные с клавиатуры. Обе функции могут работать с любым встроенным типом данных, а также с символьными строками, которые завершаются символом конца строки (‘\0’).



`printf()`

Вот прототип функции `printf()`:

```
int printf(const char * управляющая_строка, ...);
```

Функция `printf()` возвращает число выведенных символов или отрицательное значение в случае ошибки.

*Управляющая_строка*¹ состоит из элементов двух видов. Первый из них — это символы, которые предстоит вывести на экран; второй — это *спецификаторы преобразования*², которые определяют способ вывода стоящих за ними аргументов. Каждый такой спецификатор начинается со знака процента, за которым следует код формата. Аргументов должно быть ровно столько, сколько и спецификаторов, причем спецификато-

¹ Часто называется просто *форматной строкой*, *форматным стрингом* или *форматом*. — Прим. ред.

² Называются также *спецификациями формата*. — Прим. ред.

ры преобразования и аргументы должны попарно соответствовать друг другу в направлении слева направо. Например, в результате такого вызова `printf()`

```
printf("Мне нравится язык %c %s", 'C', "и к тому же очень сильно!");
```

будет выведено

```
Мне нравится язык C и к тому же очень сильно!
```

В этом примере первому спецификатору преобразования (`%c`), соответствует символ 'C', а второму (`%s`), — строка "и к тому же очень сильно!".

В функции `printf()`, как видно из табл. 8.2, имеется широкий набор спецификаторов преобразования.

Таблица 8.2. Спецификаторы преобразования для функции `printf()`

Код	Формат
<code>%a</code>	Шестнадцатеричное в виде <code>0xh.hhhhp+d</code> (только C99)
<code>%A</code>	Шестнадцатеричное в виде <code>0Xh.hhhhP+d</code> (только C99)
<code>%c</code>	Символ
<code>%d</code>	Десятичное целое со знаком
<code>%i</code>	Десятичное целое со знаком
<code>%e</code>	Экспоненциальное представление ('e' на нижнем регистре)
<code>%E</code>	Экспоненциальное представление ('E' на верхнем регистре)
<code>%f</code>	Десятичное с плавающей точкой
<code>%g</code>	В зависимости от того, какой вывод будет короче, используется <code>%e</code> или <code>%f</code>
<code>%G</code>	В зависимости от того, какой вывод будет короче, используется <code>%E</code> или <code>%F</code>
<code>%o</code>	Восьмеричное без знака
<code>%s</code>	Строка символов
<code>%u</code>	Десятичное целое без знака
<code>%x</code>	Шестнадцатеричное без знака (буквы на нижнем регистре)
<code>%X</code>	Шестнадцатеричное без знака (буквы на верхнем регистре)
<code>%p</code>	Выводит указатель
<code>%n</code>	Аргумент, соответствующий этому спецификатору, должен быть указателем на целочисленную переменную. Спецификатор позволяет сохранить в этой переменной количество записанных символов (записанных до того места, в котором находится код <code>%n</code>)
<code>%%</code>	Выводит знак <code>%</code>

Вывод символов

Для вывода отдельного символа используйте `%c`. В результате соответствующий аргумент будет выведен на экран без изменения.

Для вывода строки используйте `%s`.

Вывод чисел

Числа в десятичном формате со знаком отображаются с помощью спецификатора преобразования `%d` или `%i`. Эти спецификаторы преобразования эквивалентны; оба поддерживаются в силу сложившихся привычек программистов, например, из-за желания поддерживать те же спецификаторы, которые применяются в функции `scanf()`.

Для вывода целого значения без знака используйте `%u`.

Спецификатор преобразования `%f` дает возможность выводить числа в формате с плавающей точкой. Соответствующий аргумент должен иметь тип `double`.

Спецификаторы преобразования `%e` и `%E` в функции `printf()` позволяют отображать аргумент типа `double` в экспоненциальном формате. В общем виде числа в таком формате выглядят следующим образом:

```
x.dddddE+/-yy
```

Чтобы отобразить букву `E` в верхнем регистре, используйте спецификатор преобразования `%E`; в противном случае используйте спецификатор преобразования `%e`.

Спецификатор преобразования `%g` или `%G` указывает, что функции `printf()` необходимо выбрать один из спецификаторов: `%f` или `%e`. В результате `printf()` выберет тот спецификатор преобразования, который позволяет сделать самый короткий вывод. Если нужно, чтобы при выборе экспоненциального формата буква `E` отображалась на верхнем регистре, используйте спецификатор преобразования `%G`; в противном случае используйте спецификатор преобразования `%g`.

Применение спецификатора преобразования `%g` показано в следующей программе:

```
#include <stdio.h>

int main(void)
{
    double f;

    for(f=1.0; f<1.0e+10; f=f*10)
        printf("%g ", f);

    return 0;
}
```

В результате выполнения получится следующее:

```
1 10 100 1000 10000 100000 1e+06 1e+07 1e+08 1e+09
```

Целые числа без знака можно выводить в восьмеричном или шестнадцатеричном формате, используя спецификатор преобразования `%o` или `%x`. Так как в шестнадцатеричной системе для представления чисел от 10 до 15 используются буквы от `A` до `F`, то эти буквы можно выводить на верхнем или на нижнем регистре. Как показано ниже, в первом случае используется спецификатор преобразования `%X`, а во втором — спецификатор преобразования `%x`:

```
#include <stdio.h>

int main(void)
{
    unsigned num;

    for (num=0; num < 16; num++) {
        printf("%o ", num);
        printf("%x ", num);
        printf("%X\n", num);
    }

    return 0;
}
```

Вот что вывела эта программа:

```
0 0 0
1 1 1
2 2 2
3 3 3
4 4 4
```

```
5 5 5
6 6 6
7 7 7
10 8 8
11 9 9
12 a A
13 b B
14 c C
15 d D
16 e E
17 f F
```

Отображение адреса

Для отображения адреса используйте спецификатор преобразования `%p`. Этот спецификатор преобразования дает `printf()` указание отобразить машинный адрес в формате, совместимом с адресацией, которая используется компьютером. Следующая программа отображает адрес переменной `sample`:

```
#include <stdio.h>

int sample;

int main(void)
{
    printf("%p", &sample);

    return 0;
}
```

Спецификатор преобразования `%n`

Спецификатор `%n` довольно значительно отличается от остальных спецификаторов преобразования. Когда функция `printf()` встречает его, ничто не выводится. Вместо этого выполняется совсем другое действие: в целую переменную, указанную соответствующим аргументом функции, записывается количество выведенных символов. Другими словами, значение, которое соответствует спецификатору преобразования `%n`, должно быть указателем на переменную. После завершения вызова `printf()` в этой переменной будет храниться количество символов, выведенных до того момента, когда встретился спецификатор преобразования `%n`. Чтобы уяснить смысл этого несколько необычного спецификатора преобразования, разберитесь, как работает следующая программа:

```
#include <stdio.h>

int main(void)
{
    int count;

    printf("Это%n проверка\n", &count);
    printf("%d", count);

    return 0;
}
```

Программа отображает строку `Это проверка`, после которой появляется число `3`. Спецификатор преобразования `%n` в основном используется в программе для выполнения динамического форматирования.

Модификаторы формата

Во многих спецификаторах преобразования можно указать модификаторы¹, которые слегка меняют их значение. Например, можно указывать минимальную ширину поля, количество десятичных разрядов и выравнивание по левому краю. Модификатор формата помещают между знаком процента и кодом формата. Об этих модификаторах сейчас и пойдет речь.

Модификатор минимальной ширины поля

Целое число, расположенное между знаком % и кодом формата, играет роль *модификатора минимальной ширины поля*. Если указан модификатор минимальной ширины поля, то чтобы ширина поля вывода была не меньше указанной минимальной длины, при необходимости вывод будет дополнен пробелами. Если же выводятся строки или числа, которые длиннее указанного минимума, то они все равно будут отображаться полностью. По умолчанию для дополнения используются пробелы. А если для этого надо использовать нули, то перед модификатором ширины поля следует поместить 0. Например, %05d означает, что любое число, количество цифр которого меньше пяти, будет дополнено таким количеством нулей, чтобы число состояло из пяти цифр. В следующей программе показано, как применяется модификатор минимальной ширины поля:

```
#include <stdio.h>

int main(void)
{
    double item;

    item = 10.12304;

    printf("%f\n", item);
    printf("%10f\n", item);
    printf("%012f\n", item);

    return 0;
}
```

Вот что выводится при выполнении этой программы:

```
10.123040
 10.123040
00010.123040
```

Модификатор минимальной ширины поля чаще всего используется при создании таблиц, в которых столбцы должны быть выровнены по вертикали. Например, следующая программа выводит таблицу квадратов и кубов чисел от 1 до 19:

```
#include <stdio.h>

int main(void)
{
    int i;

    /* вывод таблицы квадратов и кубов */
    for(i=1; i<20; i++)
        printf("%8d %8d %8d\n", i, i*i, i*i*i);

    return 0;
}
```

¹ Называются также *спецификаторами*. — Прим. ред.

А вот пример полученного с ее помощью вывода:

1	1	1
2	4	8
3	9	27
4	16	64
5	25	125
6	36	216
7	49	343
8	64	512
9	81	729
10	100	1000
11	121	1331
12	144	1728
13	169	2197
14	196	2744
15	225	3375
16	256	4096
17	289	4913
18	324	5832
19	361	6859

Модификатор точности

Модификатор точности следует за модификатором минимальной ширины поля (если таковой имеется). Он состоит из точки и расположенного за ней целого числа. Значение этого модификатора зависит от типа данных, к которым его применяют.

Когда модификатор точности применяется к данным с плавающей точкой, для преобразования которых используются спецификаторы преобразования `%f`, `%e` или `%E`, то он определяет количество выводимых десятичных разрядов. Например, `%10.4f` означает, что ширина поля вывода будет не менее 10 символов, причем для десятичных разрядов будет отведено четыре позиции.

Если модификатор точности применяется к `%g` или `%G`, то он определяет количество значащих цифр.

Примененный к строкам, модификатор точности определяет максимальную длину поля. Например, `%5.7s` означает, что длина выводимой строки будет составлять минимум пять и максимум семь символов. Если строка окажется длиннее, чем максимальная длина поля, то конечные символы выводиться не будут.

Если модификатор точности применяется к целым типам, то он определяет минимальное количество цифр, которые будут выведены для каждого из чисел. Чтобы получилось требуемое количество цифр, добавляется некоторое количество ведущих нулей.

В следующей программе показано, как можно использовать модификатор точности:

```
#include <stdio.h>

int main(void)
{
    printf("%.4f\n", 123.1234567);
    printf("%3.8d\n", 1000);
    printf("%10.15s\n", "Это простая проверка.");

    return 0;
}
```

Вот что выводится при выполнении этой программы:

```
123.1235
00001000
Это простая про
```


Выравнивание вывода

По умолчанию весь вывод выравнивается по правому краю. То есть если ширина поля больше ширины выводимых данных, то эти данные располагаются по правому краю поля. Вывод по левому краю можно назначить принудительно, поместив знак минус прямо за %. Например, `%-10.2f` означает, что число с плавающей точкой и с двумя десятичными разрядами будет выровнено по левому краю 10-символьного поля.

В следующей программе показано, как применяется выравнивание по левому краю:

```
#include <stdio.h>

int main(void)
{
    printf(".....\n");
    printf("по правому краю: %8d\n", 100);
    printf(" по левому краю: %-8d\n", 100);

    return 0;
}
```

И вот что получилось:

```
.....
по правому краю:      100
по левому краю: 100
```

Обработка данных других типов

Некоторые модификаторы в вызове функции `printf()` позволяют отображать целые числа типа `short` и `long`. Такие модификаторы можно использовать для следующих спецификаторов типа: `d`, `i`, `o`, `u` и `x`. Модификатор `l` (*эль*) в вызове функции `printf()` указывает, что за ним следуют данные типа `long int`. Например, `%ld` означает, что надо вывести данные типа `long int`. После модификатора `h` функция `printf()` выведет целое значение в виде `short`. Например, `%hu` означает, что выводимые данные имеют тип `short unsigned int`.

Модификаторы `l` и `h` можно также применить к спецификатору `n`. Это делается с той целью, чтобы показать — соответствующий аргумент является указателем соответственно на длинное (`long`) или короткое (`short`) целое.

Если компилятор поддерживает обработку символов в расширенном 16-битном алфавите, добавленную Поправкой 1 от 1995 года (1995 Amendment 1), то для указания символа в расширенном 16-битном алфавите вы можете применять модификатор `l` для спецификатора преобразования `c`. Кроме того, для указания строки из символов в расширенном 16-битном алфавите можно применять модификатор `l` для спецификатора преобразования `s`.

Модификатор `L` может находиться перед спецификаторами преобразования с плавающей точкой `e`, `f` и `g`, и указывать этим, что преобразуется значение `long double`.

В Стандарте C99 вводится два новых модификатора формата: `hh` и `ll`. Модификатор `hh` можно применять для спецификаторов преобразования `d`, `i`, `o`, `u`, `x` или `n`. Он показывает, что соответствующий аргумент является значением `signed` или `unsigned char` или, в случае `n`, указателем на переменную `signed char`. Модификатор `ll` также можно применять для спецификаторов преобразования `d`, `i`, `o`, `u`, `x` или `n`. Он показывает, что соответствующий аргумент является значением `signed` или `unsigned long long int` или, в случае `n`, указателем на `long long int`. В C99 также разрешается применять `l` для спецификаторов преобразования с плавающей точкой `a`, `e`, `f` и `g`; впрочем, это не даст никакого результата.

Модификаторы * и

Для некоторых из своих спецификаторов преобразования функция `printf()` поддерживает два дополнительных модификатора: * и #.

Непосредственное расположение # перед спецификаторами преобразования `g`, `G`, `f`, `F` или `e` означает, что при выводе обязательно появится десятичная точка — даже если десятичных цифр нет. Если вы поставите # непосредственно перед `x` или `X`, то шестнадцатеричное число будет выведено с префиксом `0x`. Если # будет непосредственно предшествовать спецификатору преобразования `o`, число будет выведено с ведущим нулем. К любым другим спецификаторам преобразования модификатор # применять нельзя. (В C99 модификатор # можно применять по отношению к преобразованию `%a`; это значит, что обязательно будет выведена десятичная точка.)

Модификаторы минимальной ширины поля и точности можно передавать функции `printf()` не как константы, а как аргументы. Для этого в качестве заполнителя используйте звездочку (*). При сканировании строки формата функция `printf()` будет каждой звездочке * из этой строки ставить в соответствие очередной аргумент, причем в том порядке, в каком расположены аргументы. Например, при выполнении оператора, показанного на рис. 8.1, минимальная ширина поля будет равна 10 символам, точность — 4, а отображаться будет число 123.3.

В следующей программе показано применение обоих модификаторов # и *:

```
#include <stdio.h>

int main(void)
{
    printf("%x %#x\n", 10, 10);
    printf("%*. *f", 10, 4, 1234.34);

    return 0;
}
```

`printf("%*. *f", 10, 4, 123.3)`

Рис. 8.1. Обратите внимание на то, каким образом звездочке (*) ставится в соответствие определенное значение

scanf()

Функция `scanf()` — это программа ввода общего назначения, выполняющая ввод с консоли. Она может читать данные всех встроенных типов и автоматически преобразовывать числа в соответствующий внутренний формат. `scanf()` во многом выглядит как обратная к `printf()`. Вот прототип функции `scanf()`:

```
int scanf(const char *управляющая_строка, ...);
```

Эта функция возвращает количество тех элементов данных, которым было успешно присвоено значение. В случае ошибки `scanf()` возвращает EOF. *управляющая строка* определяет преобразование считываемых значений при записи их переменные, на которые указывают элементы списка аргументов.

Управляющая строка состоит из символов трех видов:

- спецификаторов преобразования,
- разделителей,
- символов, не являющихся разделителями.

Теперь поговорим о каждом из этих видов.

Спецификаторы преобразования

Каждый спецификатор формата ввода начинается со знака %, причем спецификаторы формата ввода сообщают функции `scanf()` тип считываемых данных. Перечень этих кодов (т.е. литер-спецификаторов) приведен в табл. 8.3. Спецификаторам преобразования в порядке слева направо ставятся в соответствие элементы списка аргументов. Рассмотрим некоторые примеры.

Ввод чисел

Для чтения целого числа используйте спецификатор преобразования `%d` или `%i`. А для чтения числа с плавающей точкой, представленного в стандартном или экспоненциальном виде, используйте спецификатор преобразования `%e`, `%f` или `%g`. (Кроме того, для чтения числа с плавающей точкой стандарт C99 разрешает использовать также спецификатор преобразования `%a`.)

Таблица 8.3. Спецификаторы преобразования для функции `scanf()`

Код	Значение
<code>%a</code>	Читает значение с плавающей точкой (только C99)
<code>%c</code>	Читает одиночный символ
<code>%d</code>	Читает десятичное целое число
<code>%i</code>	Читает целое число как в десятичном, так и восьмеричном или шестнадцатеричном формате
<code>%e</code>	Читает число с плавающей точкой
<code>%f</code>	Читает число с плавающей точкой
<code>%g</code>	Читает число с плавающей точкой
<code>%o</code>	Читает восьмеричное число
<code>%s</code>	Читает строку
<code>%x</code>	Читает шестнадцатеричное число
<code>%p</code>	Читает указатель
<code>%n</code>	Принимает целое значение, равное количеству уже считанных символов
<code>%u</code>	Читает десятичное целое число без знака
<code>%[]</code>	Читает набор сканируемых символов
<code>%%</code>	Читает знак процента

Функцию `scanf()` можно использовать для чтения целых значений в восьмеричной или шестнадцатеричной форме, применяя для этого соответственно команды форматирования `%o` и `%x`, последняя из которых может быть как на верхнем, так и на нижнем регистре. Когда вводятся шестнадцатеричные числа, то буквы от A до F, представляющие шестнадцатеричные цифры, должны быть на том же самом регистре, что и литер-спецификатор. Следующая программа читает восьмеричное и шестнадцатеричное число:

```
#include <stdio.h>

int main(void)
{
    int i, j;

    scanf("%o%x", &i, &j);
    printf("%o %x", i, j);

    return 0;
}
```

Функция `scanf()` прекращает чтение числа тогда, когда встречается первый нечисловой символ.

Ввод целых значений без знака

Для ввода целого значения без знака используйте спецификатор формата `%u`. Например, операторы

```
unsigned num;
scanf("%u", &num);
```

выполняют считывание целого числа без знака и присваивают его переменной `num`.

Чтение одиночных символов с помощью `scanf()`

Как уже говорилось в этой главе, одиночные символы можно прочесть с помощью функции `getchar()` или какой-либо функции, родственной с ней. Для той же цели можно использовать также вызов функции `scanf()` со спецификатором формата `%c`. Но, как и большинство реализаций `getchar()`, функция `scanf()` при использовании спецификатора преобразования `%c` обычно будет выполнять построчно буферизованный ввод. В интерактивной среде такая ситуация вызывает определенные трудности.

При чтении одиночного символа символы разделителей читаются так же, как и любой другой символ, хотя при чтении данных других типов разделители интерпретируются как разделители полей. Например, при вводе с входного потока “`x y`” фрагмент кода

```
scanf("%c%c%c", &a, &b, &c);
```

помещает символ `x` в `a`, пробел — в `b`, а символ `y` — в `c`.

Чтение строк

Для чтения из входного потока строки можно использовать функцию `scanf()` со спецификатором преобразования `%s`. Использование спецификатора преобразования `%s` заставляет `scanf()` читать символы до тех пор, пока не встретится какой-либо разделитель. Читаемые символы помещаются в символьный массив, на который указывает соответствующий аргумент, а после введенных символов еще добавляется символ конца строки (‘`0`’). Что касается `scanf()`, то таким разделителем может быть пробел, разделитель строк, табуляция, вертикальная табуляция или подача страницы. В отличие от `gets()`, которая читает строку, пока не будет нажата клавиша <ENTER>, `scanf()` читает строку до тех пор, пока не встретится первый разделитель. Это означает, что `scanf()` нельзя использовать для чтения строки “это испытание”, потому что после пробела процесс чтения прекратится. Чтобы увидеть, как действует спецификатор `%s`, попробуйте при выполнении этой программы ввести строку “привет всем”:

```
#include <stdio.h>

int main(void)
{
    char str[80];

    printf("Введите строку: ");
    scanf("%s", str);
    printf("Вот ваша строка: %s", str);

    return 0;
}
```

Программа выведет только часть строки, то есть слово “привет”.

Ввод адреса

Для ввода какого-либо адреса памяти используйте спецификатор преобразования `%p`. Этот спецификатор преобразования заставляет функцию `scanf()` читать адрес в том формате, который определен архитектурой центрального процессора. Например, следующая программа вначале вводит адрес, а затем отображает то, что находится в памяти по этому адресу:

```
#include <stdio.h>

int main(void)
{
    char *p;

    printf("Введите адрес: ");
    scanf("%p", &p);
    printf("По адресу %p находится %c\n", p, *p);

    return 0;
}
```

Спецификатор `%n`

Спецификатор `%n` указывает, что `scanf()` должна поместить количество символов, считанных (до того момента, когда встретился `%n`) из входного потока в целую переменную, указанную соответствующим аргументом.

Использование набора сканируемых символов

Функция `scanf()` поддерживает спецификатор формата общего назначения, называемый набором сканируемых символов (`scanset`). *Набор сканируемых символов* представляет собой множество символов. Когда `scanf()` обрабатывает такое множество, то вводит только те символы, которые входят в набор сканируемых символов. Читаемые символы будут помещаться в массив символов, который указан аргументом, соответствующим набору сканируемых символов. Этот набор определяется следующим образом: все те символы, которые предстоит сканировать, помещают в квадратные скобки. Непосредственно перед открывающей квадратной скобкой должен находиться знак `%`. Например, следующий набор сканируемых символов дает указание `scanf()` сканировать только символы `X`, `Y` и `Z`:

```
%[XYZ]
```

При использовании набора сканируемых символов функция `scanf()` продолжает читать символы, помещая их в соответствующий массив символов, пока не встретится символ, не входящий в этот набор. При возвращении из `scanf()` в массиве символов будет находиться строка, состоящая из считанных символов, причем эта строка будет заканчиваться символом конца строки. Чтобы увидеть, как это все работает, запустите следующую программу:

```
#include <stdio.h>

int main(void)
{
    int i;
    char str[80], str2[80];

    scanf("%d%[abcdefg]%s", &i, str, str2);
    printf("%d %s %s", i, str, str2);

    return 0;
}
```

Введите `123abcdtye`, а затем нажмите клавишу `<ENTER>`. После этого программа выведет `123 abcd tye`. Так как в данном случае `'t'` не входит в набор сканируемых символов, то `scanf()` прекратила чтение символов в переменную `str` сразу после того, как встретился символ `'t'`. Оставшиеся символы были помещены в переменную `str2`.

Кроме того, можно указать набор сканируемых символов, работающий с точностью до наоборот; тогда первым символом в таком наборе должен быть `^`. Этот символ дает указание `scanf()` принимать любой символ, который *не* входит в набор сканируемых символов.

В большинстве реализаций для указания диапазона можно использовать дефис. Например, указанный ниже набор сканируемых символов дает функции `scanf()` указание принимать символы от `A` до `Z`:

```
%[A-Z]
```

Следует обратить внимание на такой важный момент: набор сканируемых символов чувствителен к регистру букв. Если нужно сканировать буквы и на верхнем, и на нижнем регистре, то их надо указывать отдельно для каждого регистра.

Пропуск лишних разделителей

Разделитель в управляющей строке дает `scanf()` указание пропустить в потоке ввода один или несколько начальных разделителей. Разделителями являются пробелы, табуляции, вертикальные табуляции, подачи страниц и разделители строк. В сущности, один разделитель в управляющей строке заставляет `scanf()` читать, но не сохранять любое количество (в том числе и нулевое) разделителей, которые находятся перед первым символом, не являющимся разделителем.

Символы в управляющей строке, не являющиеся разделителями

Если в управляющей строке находится символ, не являющийся разделителем, то функция `scanf()` прочитает символ из входного потока, проверит, совпадает ли прочитанный символ с указанным в управляющей строке, и в случае совпадения пропустит прочитанный символ. Например, `"%d,%d"` заставляет `scanf()` прочитывать целое значение, прочитав запятую и пропустить ее (если это была запятая!), а затем прочи-

тать следующее целое значение. Если же указанный символ во входном потоке не будет найден, то `scanf()` завершится. Когда нужно прочитать и отбросить знак процента, то в управляющей строке следует указать `%%`.

Функции `scanf()` необходимо передавать адреса

Для всех переменных, которые должны получить значения с помощью `scanf()`, должны быть переданы адреса. Это означает, что все аргументы должны быть указателями. Вспомните, что именно так в C создается вызов по ссылке и именно тогда функция может изменить содержимое аргумента. Например, для считывания целого значения в переменную `count` можно использовать такой вызов функции `scanf()`:

```
scanf("%d", &count);
```

Строки будут читаться в символьные массивы, а имя массива без индекса является адресом первого его элемента. Таким образом, чтобы прочитать строку в символьный массив, можно использовать оператор

```
scanf("%s", str);
```

В этом случае `str` является указателем, и потому перед ним не нужно ставить оператор `&`.

Модификаторы формата

Как и `printf()`, функция `scanf()` дает возможность модифицировать некоторое число своих спецификаторов формата. В спецификаторах формата можно указать модификатор максимальной длины поля. Это целое число, расположенное между `%` и спецификатором формата; оно ограничивает число символов, считываемых из этого поля. Например, чтобы считывать в переменную `str` не более 20 символов, пишите

```
scanf("%20s", str);
```

Если поток ввода содержит больше 20 символов, то при следующем вызове функций ввода считывание начнется после того места, где оно закончилось при предыдущем вызове. Например, если вы в ответ на вызов `scanf()` из этого примера введете

ABCDEFGHIJKLMNOPRSTUVWXYZ

то в `str` из-за спецификатора максимальной ширины поля будет помещено только 20 символов, то есть символы вплоть до T. Это значит, что оставшиеся символы UVWXYZ пока еще не прочитаны. При следующем вызове `scanf()`, например при выполнении оператора

```
scanf("%s", str);
```

в `str` будут помещены буквы UVWXYZ. Ввод из поля может завершиться и до того, как будет достигнута максимальная длина поля — если встретится разделитель. В таком случае `scanf()` переходит к следующему полю.

Чтобы прочитать длинное целое, перед спецификатором формата поместите `l` (эль). А для чтения короткого целого значения перед спецификатором формата следует поместить `h`. Эти модификаторы можно использовать со следующими кодами форматов: `d`, `i`, `o`, `u`, `x` и `n`.

По умолчанию спецификаторы `f`, `e` и `g` дают `scanf()` указание присваивать данные переменной типа `float`. Если перед одним из этих спецификаторов будет помещен `l` (эль), то `scanf()` будет присваивать данные переменной типа `double`. Использование `L` дает `scanf()` указание, чтобы переменная, принимающая данные, имела тип `long double`.

Если в компиляторе предусмотрена обработка *двухбайтовых символов*¹, добавленных в язык C Поправкой 1 от 1995 года, то модификатор `l` можно также использовать с такими кодами формата, как `c` и `s`. `l` непосредственно перед `c` является признаком указателя на объект типа `wchar_t`. А `l` непосредственно перед `s` — признак указателя на массив элементов типа `wchar_t`. Кроме того, `l` также применяется для модификации набора сканируемых символов, чтобы этот набор можно было использовать для двухбайтовых символов.

В Стандарте C99, кроме перечисленных, предусмотрены также модификаторы `ll` и `hh`, последний из которых можно применять к спецификаторам `d`, `i`, `o`, `u`, `x` или `n`. Он является признаком того, что соответствующий аргумент является указателем на значение типа `signed` или `unsigned char`. Кроме того, к спецификаторам `d`, `i`, `o`, `u`, `x` и `n` можно применять и `ll`. Этот спецификатор является признаком того, что соответствующий аргумент является указателем на значение типа `signed (или unsigned) long long int`.

На заметку

В C99 для функции `scanf()` имеются еще и другие модификаторы типа; о них рассказывается в части II.

Подавление ввода

`scanf()` может прочитать поле, но не присваивать прочитанное значение никакой переменной; для этого надо перед литерой-спецификатором формата поля поставить звездочку, `*`. Например, когда выполняется оператор

```
scanf("%d%c%d", &x, &y);
```

можно ввести пару координат `10,10`. Запятая будет прочитана правильно, но ничему не будет присвоена. Подавление присвоения особенно полезно тогда, когда нужно обработать только часть того, что вводится.

¹ Называются также *символами в расширенном 16-битном алфавите* или *символами уникода*. (Unicode (уникод) — 16-битовый стандарт кодирования символов, позволяющий представлять алфавиты всех существующих в мире языков.) — *Прим. ред.*

Полный
справочник по



Глава 9

Файловый ввод/вывод

В этой главе описана работа с файловой системой в языке C. Как уже говорилось в главе 8, в языке C система ввода/вывода реализуется с помощью библиотечных функций, а не ключевых слов. Благодаря этому система ввода/вывода является очень мощной и гибкой. Например, во время работы с файлами данные могут передаваться или в своем внутреннем двоичном представлении или в текстовом формате, то есть в более удобочитаемом виде. Это облегчает задачу создания файлов в нужном формате.



Файловый ввод/вывод в C и C++

Так как C является фундаментом C++, то иногда возникает путаница в отношении его файловой системы с аналогичной системой C++. Во-первых, C++ поддерживает всю файловую систему C. Таким образом, при перемещении более старого C-кода в C++ нет необходимости менять все процедуры ввода/вывода. Во-вторых, следует иметь в виду, что в C++ определена своя собственная, объектно-ориентированная система ввода/вывода, в которую входят как функции, так и операторы ввода/вывода. В системе ввода/вывода C++ полностью поддерживаются все возможности аналогичной системы C и это делает излишней файловую систему языка C. Вообще говоря, при написании программ на языке C++ обычно более удобно использовать именно его систему ввода/вывода, но, если необходимо воспользоваться файловой системой языка C, то это также вполне возможно.



Файловый ввод/вывод в стандартном C и UNIX

Первоначально язык C был реализован в операционной системе UNIX. Как таковые, ранние версии C (да и многие нынешние) поддерживают набор функций ввода/вывода, совместимый с UNIX. Этот набор иногда называют *UNIX-подобной системой ввода/вывода* или *небуферизованной системой ввода/вывода*. Однако когда C был стандартизован, то UNIX-подобные функции в него не вошли — в основном из-за того, что оказались лишними. Кроме того, UNIX-подобная система может оказаться неподходящей для некоторых сред, которые могут поддерживать язык C, но не эту систему ввода/вывода.

В этой главе говорится только о тех функциях ввода/вывода, которые определены в стандарте C. В предыдущих изданиях этой книги немного говорилось и о UNIX-подобной файловой системе. Но в течение того времени, которое прошло с выхода последнего издания, число случаев использования стандартных функций ввода/вывода устойчиво росло, а UNIX-подобных функций — устойчиво падало. И теперь большинство программистов пользуются стандартными функциями — эти функции можно переносить во все среды (и даже в C++). А тем программистам, которым нужно пользоваться UNIX-подобными функциями, приходится обращаться к документации по имеющимся у них компиляторам.



Потоки и файлы

Перед тем как начать изучение файловой системы языка C, необходимо уяснить, в чем разница между *потоками* и *файлами*. В системе ввода/вывода C для программ поддерживается единый интерфейс, не зависящий от того, к какому конкретному устройству осуществляется доступ. То есть в этой системе между программой и устройством находится нечто более общее, чем само устройство. Такое обобщенное устройство ввода или вывода (устройство более высокого уровня абстракции. — *Прим. ред.*) назы-

вается *поток*ом, в то время как конкретное устройство называется *файлом*. (Впрочем, файл — тоже понятие абстрактное. — *Прим. ред.*) Очень важно понимать, каким образом происходит взаимодействие потоков и файлов.

Потоки

Файловая система языка С предназначена для работы с самыми разными устройствами, в том числе терминалами, дисковыми и накопителями на магнитной ленте. Даже если какое-то устройство сильно отличается от других, буферизованная файловая система все равно представит его в виде логического устройства, которое называется потоком. Все потоки ведут себя похожим образом. И так как они в основном не зависят от физических устройств, то та же функция, которая выполняет запись в дисковый файл, может ту же операцию выполнять и на другом устройстве, например, на консоли. Потоки бывают двух видов: текстовые и двоичные.

Текстовые потоки

Текстовый поток — это последовательность символов. В стандарте С считается, что текстовый поток организован в виде строк, каждая из которых заканчивается символом новой строки. Однако в конце последней строки этот символ не является обязательным. В текстовом потоке по требованию базовой среды могут происходить определенные преобразования символов. Например, символ новой строки может быть заменен парой символов — возврата каретки и перевода строки. Поэтому может и не быть однозначного соответствия между символами, которые пишутся (читаются), и теми, которые хранятся во внешнем устройстве. Кроме того, количество тех символов, которые пишутся (читаются), и тех, которые хранятся во внешнем устройстве, может также не совпадать из-за возможных преобразований.

Двоичные потоки

Двоичный поток — это последовательность байтов, которая взаимно однозначно соответствует байтам на внешнем устройстве, причем никакого преобразования символов не происходит. Кроме того, количество тех байтов, которые пишутся (читаются), и тех, которые хранятся на внешнем устройстве, одинаково. Однако в конце двоичного потока может добавляться определяемое приложением количество нулевых байтов. Такие нулевые байты, например, могут использоваться для заполнения свободного места в блоке памяти незначащей информацией, чтобы она в точности заполнила сектор на диске.

Файлы

В языке С файлом может быть все что угодно, начиная с дискового файла и заканчивая терминалом или принтером. Поток связывают с определенным файлом, выполняя операцию *открытия*. Как только файл открыт, можно проводить обмен информацией между ним и программой.

Но не у всех файлов одинаковые возможности. Например, к дисковому файлу прямой доступ возможен, в то время как к некоторым принтерам — нет. Таким образом, мы пришли к одному важному принципу, относящемуся к системе ввода/вывода языка С: все потоки одинаковы, а файлы — нет.

Если файл может поддерживать *запросы на местоположение (указатель текущей позиции)*, то при открытии такого файла *указатель текущей позиции в файле* устанавливается в начало. При чтении из файла (или записи в него) каждого символа указатель текущей позиции увеличивается, обеспечивая тем самым продвижение по файлу.

Файл отсоединяется от определенного потока (т.е. разрывается связь между файлом и потоком) с помощью операции *закрытия*. При закрытии файла, открытого с

целью вывода, содержимое (если оно есть) связанного с ним потока записывается на внешнее устройство. Этот процесс, который обычно называют *дозаписью*¹ потока, гарантирует, что никакая информация случайно не останется в буфере диска. Если программа завершает работу нормально, т.е. либо `main()` возвращает управление операционной системе, либо вызывается `exit()`, то все файлы закрываются автоматически. В случае аварийного завершения работы программы, например, в случае краха или завершения путем вызова `abort()`, файлы не закрываются.

У каждого потока, связанного с файлом, имеется управляющая структура, содержащая информацию о файле; она имеет тип `FILE`. В этом *блоке управления файлом*² никогда ничего не меняйте³.

Если вы новичок в программировании, то разграничение потоков и файлов может показаться излишним или даже “заумным”. Однако надо помнить, что основная цель такого разграничения — это обеспечить единый интерфейс. Для выполнения всех операций ввода/вывода следует использовать только понятия потоков и применять всего лишь одну файловую систему. Ввод или вывод от каждого устройства автоматически преобразуется системой ввода/вывода в легко управляемый поток.



Основы файловой системы

Файловая система языка C состоит из нескольких взаимосвязанных функций. Самые распространенные из них показаны в табл. 9.1. Для их работы требуется заголовок `<stdio.h>`.

Таблица 9.1. Часто используемые функции файловой системы C

Имя	Что делает
<code>fopen()</code>	Открывает файл
<code>fclose()</code>	Закрывает файл
<code>putc()</code>	Записывает символ в файл
<code>fputc()</code>	То же, что и <code>putc()</code>
<code>getc()</code>	Читает символ из файла
<code>fgetc()</code>	То же, что и <code>getc()</code>
<code>fgets()</code>	Читает строку из файла
<code>fputs()</code>	Записывает строку в файл
<code>fseek()</code>	Устанавливает указатель текущей позиции на определенный байт файла
<code>ftell()</code>	Возвращает текущее значение указателя текущей позиции в файле
<code>fprintf()</code>	Для файла то же, что <code>printf()</code> для консоли
<code>fscanf()</code>	Для файла то же, что <code>scanf()</code> для консоли
<code>feof()</code>	Возвращает значение <code>true</code> (истина), если достигнут конец файла
<code>ferror()</code>	Возвращает значение <code>true</code> , если произошла ошибка
<code>rewind()</code>	Устанавливает указатель текущей позиции в начало файла
<code>remove()</code>	Стирает файл
<code>fflush()</code>	Дозапись потока в файл

¹ Или *принудительным освобождением (содержимого) буфера*. — Прим. ред.

² *Блок управления файлом* — небольшой блок памяти, временно выделенный операционной системой для хранения информации о файле, который был открыт для использования. Блок управления файлом обычно содержит информацию об идентификаторе файла, его расположении на диске и указателе текущей позиции в файле. — Прим. ред.

³ Если, конечно, вы не разрабатываете систему ввода-вывода. — Прим. ред.

Заголовок `<stdio.h>` предоставляет прототипы функций ввода/вывода и определяет следующие три типа: `size_t`, `fpos_t` и `FILE`. `size_t` и `fpos_t` представляют собой определенные разновидности такого типа, как целое без знака. А о третьем типе, `FILE`, рассказывается в следующем разделе.

Кроме того, в `<stdio.h>` определяется несколько макросов. Из них к материалу этой главы относятся `NULL`, `EOF`, `FOPEN_MAX`, `SEEK_SET`, `SEEK_CUR` и `SEEK_END`. Макрос `NULL` определяет пустой (`null`) указатель. Макрос `EOF`, часто определяемый как `-1`, является значением, возвращаемым тогда, когда функция ввода пытается выполнить чтение после конца файла. `FOPEN_MAX` определяет целое значение, равное максимальному числу одновременно открытых файлов. Другие макросы используются вместе с `fseek()` — функцией, выполняющей операции прямого доступа к файлу.

Указатель файла

Указатель файла — это то, что соединяет в единое целое всю систему ввода/вывода языка C. *Указатель файла* — это указатель на структуру типа `FILE`. Он указывает на структуру, содержащую различные сведения о файле, например, его имя, статус и указатель текущей позиции в начале файла. В сущности, указатель файла определяет конкретный файл и используется соответствующим потоком при выполнении функций ввода/вывода. Чтобы выполнять в файлах операции чтения и записи, программы должны использовать указатели соответствующих файлов. Чтобы объявить переменную-указатель файла, используйте такого рода оператор:

```
FILE *fp;
```

Открытие файла

Функция `fopen()` открывает поток и связывает с этим потоком определенный файл. Затем она возвращает указатель этого файла. Чаще всего (а также в оставшейся части этой главы) под файлом подразумевается дисковый файл. Прототип функции `fopen()` такой:

```
FILE *fopen(const char *имя_файла, const char *режим);
```

где *имя_файла* — это указатель на строку символов, представляющую собой допустимое имя файла, в которое также может входить спецификация пути к этому файлу. Строка, на которую указывает *режим*, определяет, каким образом файл будет открыт. В табл. 9.2 показано, какие значения строки *режим* являются допустимыми. Строки, подобные “r+b” могут быть представлены и в виде “rb+”.

Как уже упоминалось, функция `fopen()` возвращает указатель файла. Никогда не следует изменять значение этого указателя в программе. Если при открытии файла происходит ошибка, то `fopen()` возвращает пустой (`null`) указатель.

В следующем коде функция `fopen()` используется для открытия файла по имени `TEST` для записи.

```
FILE *fp;  
fp = fopen("test", "w");
```

Таблица 9.2. Допустимые значения режим

Режим	Что означает
r	Открыть текстовый файл для чтения
w	Создать текстовый файл для записи
a	Добавить в конец текстового файла
rb	Открыть двоичный файл для чтения
wb	Создать двоичный файл для записи

Режим	Что означает
ab	Добавить в конец двоичного файла
r+	Открыть текстовый файл для чтения/записи
w+	Создать текстовый файл для чтения/записи
a+	Добавить в конец текстового файла или создать текстовый файл для чтения/записи
r+b	Открыть двоичный файл для чтения/записи
w+b	Создать двоичный файл для чтения/записи
a+b	Добавить в конец двоичного файла или создать двоичный файл для чтения/записи

Хотя предыдущий код технически правильный, но его обычно пишут немного по-другому:

```
FILE *fp;

if ((fp = fopen("test", "w"))==NULL) {
    printf("Ошибка при открытии файла.\n");
    exit(1);
}
```

Этот метод помогает при открытии файла обнаружить любую ошибку, например, защиту от записи или полный диск, причем обнаружить еще до того, как программа попытается в этот файл что-либо записать. Вообще говоря, всегда нужно вначале получить подтверждение, что функция `fopen()` выполнилась успешно, и лишь затем выполнять с файлом другие операции.

Хотя название большинства файловых режимов объясняет их смысл, однако не помещает сделать некоторые дополнения. Если попытаться открыть файл только для чтения, а он не существует, то работа `fopen()` завершится отказом. А если попытаться открыть файл в режиме дозаписи, а сам этот файл не существует, то он просто будет создан. Более того, если файл открыт в режиме дозаписи, то все новые данные, которые записываются в него, будут добавляться в конец файла. Содержимое, которое хранилось в нем до открытия (если только оно было), изменено не будет. Далее, если файл открывают для записи, но выясняется, что он не существует, то он будет создан. А если он существует, то содержимое, которое хранилось в нем до открытия, будет утеряно, причем будет создан новый файл. Разница между режимами `r+` и `w+` состоит в том, что если файл не существует, то в режиме открытия `r+` он создан не будет, а в режиме `w+` все произойдет наоборот: файл будет создан! Более того, если файл уже существует, то открытие его в режиме `w+` приведет к утрате его содержимого, а в режиме `r+` оно останется нетронутым.

Из табл. 9.2 видно, что файл можно открыть либо в одном из текстовых, либо в одном из двоичных режимов. В большинстве реализаций в текстовых режимах каждая комбинация кодов возврата каретки (ASCII 13) и конца строки (ASCII 10) преобразуется при вводе в символ новой строки. При выводе же происходит обратный процесс: символы новой строки преобразуются в комбинацию кодов возврата каретки (ASCII 13) и конца строки (ASCII 10). В двоичных режимах такие преобразования не выполняются.

Максимальное число одновременно открытых файлов определяется `FOPEN_MAX`. Это значение не меньше 8, но чему оно точно равняется — это должно быть написано в документации по компилятору.

Заккрытие файла

Функция `fclose()` закрывает поток, который был открыт с помощью вызова `fopen()`. Функция `fclose()` записывает в файл все данные, которые еще оставались в дисковом буфере, и проводит, так сказать, официальное закрытие файла на уровне

операционной системы. Отказ при закрытии потока влечет всевозможные неприятности, включая потерю данных, испорченные файлы и возможные периодические ошибки в программе. Функция `fclose()` также освобождает блок управления файлом, связанный с этим потоком, давая возможность использовать этот блок снова. Так как количество одновременно открытых файлов ограничено, то, возможно, придется закрывать один файл, прежде чем открывать другой.

Прототип функции `fclose()` такой:

```
int fclose(FILE *уф);
```

где *уф* — указатель файла, возвращенный в результате вызова `fopen()`. Возвращение нуля означает успешную операцию закрытия. В случае же ошибки возвращается `EOF`. Чтобы точно узнать, в чем причина этой ошибки, можно использовать стандартную функцию `ferror()` (о которой вскоре пойдет речь). Обычно отказ при выполнении `fclose()` происходит только тогда, когда диск был преждевременно удален (стерт) с дисководы или на диске не осталось свободного места.

Запись символа

В системе ввода/вывода языка C определяются две эквивалентные функции, предназначенные для вывода символов: `putc()` и `fputc()`. (На самом деле `putc()` обычно реализуется в виде макроса.) Две идентичные функции имеются просто потому, чтобы сохранять совместимость со старыми версиями C. В этой книге используется `putc()`, но применение `fputc()` также вполне возможно.

Функция `putc()` записывает символы в файл, который с помощью `fopen()` уже открыт в режиме записи. Прототип этой функции следующий:

```
int putc(int ch, FILE *уф);
```

где *уф* — это указатель файла, возвращенный функцией `fopen()`, а *ch* — выводимый символ. Указатель файла сообщает `putc()`, в какой именно файл следует записывать символ. Хотя *ch* и определяется как `int`, однако записывается только младший байт.

Если функция `putc()` выполнилась успешно, то возвращается записанный символ. В противном же случае возвращается `EOF`.

Чтение символа

Для ввода символа также имеются две эквивалентные функции: `getc()` и `fgetc()`. Обе определяются для сохранения совместимости со старыми версиями C. В этой книге используется `getc()` (которая обычно реализуется в виде макроса), но если хотите, применяйте `fgetc()`.

Функция `getc()` записывает символы в файл, который с помощью `fopen()` уже открыт в режиме для чтения. Прототип этой функции следующий:

```
int getc(FILE *уф);
```

где *уф* — это указатель файла, имеющий тип `FILE` и возвращенный функцией `fopen()`. Функция `getc()` возвращает целое значение, но символ находится в младшем байте. Если не произошла ошибка, то старший байт (байты) будет обнулен.

Если достигнут конец файла, то функция `getc()` возвращает `EOF`. Поэтому, чтобы прочитать символы до конца текстового файла, можно использовать следующий код:

```
do {  
    ch = getc(fp);  
} while(ch!=EOF);
```

Однако `getc()` возвращает `EOF` и в случае ошибки. Для определения того, что же на самом деле произошло, можно использовать `ferror()`.

Использование fopen(), getc(), putc() и fclose()

Функции fopen(), getc(), putc() и fclose() — это минимальный набор функций для операций с файлами. Следующая программа, KТОD, представляет собой простой пример, в котором используются только функции putc(), fopen() и fclose(). В этой программе символы считываются с клавиатуры и записываются в дисковый файл до тех пор, пока пользователь не введет знак доллара. Имя файла определяется в командной строке. Например, если вызвать программу KТОD, введя в командной строке KТОD TEST, то строки текста будут вводиться в файл TEST.

```
/* KТОD: программа ввода с клавиатуры на диск. */
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
    FILE *fp;
    char ch;

    if(argc!=2) {
        printf("Вы забыли ввести имя файла.\n");
        exit(1);
    }

    if((fp=fopen(argv[1], "w"))==NULL) {
        printf("Ошибка при открытии файла.\n");
        exit(1);
    }

    do {
        ch = getchar();
        putc(ch, fp);
    } while (ch != '$');

    fclose(fp);

    return 0;
}
```

Программа DTOS, являющаяся дополнением к программе KТОD, читает любой текстовый файл и выводит его содержимое на экран.

```
/* DTOS: программа, которая читает файлы и выводит их на экран. */
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
    FILE *fp;
    char ch;

    if(argc!=2) {
        printf("Вы забыли ввести имя файла.\n");
        exit(1);
    }

    if((fp=fopen(argv[1], "r"))==NULL) {
        printf("Ошибка при открытии файла.\n");
    }
```



```

    exit(1);
}

ch = getc(fp); /* чтение одного символа */

while (ch!=EOF) {
    putchar(ch); /* вывод на экран */
    ch = getc(fp);
}

fclose(fp);

return 0;
}

```

Испытывая эти две программы, вначале с помощью KTOD создайте текстовый файл, а затем с помощью DTOS прочитайте его содержимое.

Использование feof()

Как уже говорилось, если достигнут конец файла, то `getc()` возвращает EOF. Однако проверка значения, возвращенного `getc()`, возможно, не является наилучшим способом узнать, достигнут ли конец файла. Во-первых, файловая система языка C может работать как с текстовыми, так и с двоичными файлами. Когда файл открывается для двоичного ввода, то может быть прочитано целое значение, которое, как выяснится при проверке, равняется EOF. В таком случае программа ввода сообщит о том, что достигнут конец файла, чего на самом деле может и не быть. Во-вторых, функция `getc()` возвращает EOF и в случае отказа, а не только тогда, когда достигнут конец файла. Если использовать только возвращаемое значение `getc()`, то невозможно определить, что же на самом деле произошло. Для решения этой проблемы в C имеется функция `feof()`, которая определяет, достигнут ли конец файла. Прототип функции `feof()` такой:

```
int feof(FILE *уф);
```

Если достигнут конец файла, то `feof()` возвращает *true* (истина); в противном же случае эта функция возвращает нуль. Поэтому следующий код будет читать двоичный файл до тех пор, пока не будет достигнут конец файла:

```
while(!feof(fp)) ch = getc(fp);
```

Ясно, что этот метод можно применять как к двоичным, так и к текстовым файлам.

В следующей программе, которая копирует текстовые или двоичные файлы, имеется пример применения `feof()`. Файлы открываются в двоичном режиме, а затем `feof()` проверяет, не достигнут ли конец файла.

```

/* Копирование файла. */
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
    FILE *in, *out;
    char ch;
    if(argc!=3) {
        printf("Вы забыли ввести имя файла.\n");
        exit(1);
    }

    if((in=fopen(argv[1], "rb"))==NULL) {

```

```

    printf("Нельзя открыть исходный файл.\n");
    exit(1);
}

if((out=fopen(argv[2], "wb")) == NULL) {
    printf("Нельзя открыть файл результатов.\n");
    exit(1);
}
/* Именно этот код копирует файл. */
while(!feof(in)) {
    ch = getc(in);
    if(!feof(in)) putc(ch, out);
}

fclose(in);
fclose(out);

return 0;
}

```

Ввод/вывод строк: fputs() и fgets()

Кроме `getc()` и `putc()`, в языке C также поддерживаются родственные им функции `fgets()` и `fputs()`. Первая из них читает строки символов из файла на диске, а вторая записывает строки такого же типа в файл, тоже находящийся на диске. Эти функции работают почти как `putc()` и `getc()`, но читают и записывают не один символ, а целую строку. Прототипы функций `fgets()` и `fputs()` следующие:

```

int fputs(const char *cmp, FILE *yf);
char *fgets(char *cmp, int длина, FILE *yf);

```

Функция `fputs()` пишет в определенный поток строку, на которую указывает `cmp`. В случае ошибки эта функция возвращает EOF.

Функция `fgets()` читает из определенного потока строку, и делает это до тех пор, пока не будет прочитан символ новой строки или количество прочитанных символов не станет равным *длина*-1. Если был прочитан разделитель строк, он записывается в строку, чем функция `fgets()` отличается от функции `gets()`. Полученная в результате строка будет оканчиваться символом конца строки ('0'). При успешном завершении работы функция возвращает `cmp`, а в случае ошибки — пустой указатель (null).

В следующей программе показано использование функции `fputs()`. Она читает строки с клавиатуры и записывает их в файл, который называется TEST. Чтобы завершить выполнение программы, введите пустую строку. Так как функция `gets()` не записывает разделитель строк, то его приходится специально вставлять перед каждой строкой, записываемой в файл; это делается для того, чтобы файл было легче читать:

```

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

int main(void)
{
    char str[80];
    FILE *fp;

    if((fp = fopen("TEST", "w"))==NULL) {
        printf("Ошибка при открытии файла.\n");
        exit(1);
    }
}

```

```

do {
    printf("Введите строку (пустую — для выхода):\n");
    gets(str);
    strcat(str, "\n"); /* добавление разделителя строк */
    fputs(str, fp);
} while(*str!='\n');

return 0;
}

```

Функция rewind()

Функция `rewind()` устанавливает указатель текущей позиции в файле на начало файла, указанного в качестве аргумента этой функции. Иными словами, функция `rewind()` выполняет “перемотку” (`rewind`) файла. Вот ее прототип:

```
void rewind(FILE *уф);
```

где *уф* — это допустимый указатель файла.

Чтобы познакомиться с `rewind()`, изменим программу из предыдущего раздела таким образом, чтобы она отображала содержимое файла сразу после его создания. Чтобы выполнить отображение, программа после завершения ввода “перематывает” файл, а затем с помощью `fback()` читает его с самого начала. Обратите внимание, что сейчас файл необходимо открыть в режиме чтения/записи, используя в качестве аргумента, задающего режим, строку “w+”.

```

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

int main(void)
{
    char str[80];
    FILE *fp;

    if((fp = fopen("TEST", "w+"))==NULL) {
        printf("Ошибка при открытии файла.\n");
        exit(1);
    }

    do {
        printf("Введите строку (пустую — для выхода):\n");
        gets(str);
        strcat(str, "\n"); /* ввод разделителя строк */
        fputs(str, fp);
    } while(*str!='\n');

    /* теперь выполняется чтение и отображение файла */
    rewind(fp); /* установить указатель
                текущей позиции на начало файла */
    while(!feof(fp)) {
        fgets(str, 79, fp);
        printf(str);
    }

    return 0;
}

```

Функция `ferror()`

Функция `ferror()` определяет, произошла ли ошибка во время выполнения операции с файлом. Прототип этой функции следующий:

```
int ferror(FILE *уф);
```

где `уф` — допустимый указатель файла. Она возвращает значение *true* (истина), если при последней операции с файлом произошла ошибка; в противном же случае она возвращает *false* (ложь). Так как при любой операции с файлом устанавливается свое условие ошибки, то после каждой такой операции следует сразу вызывать `ferror()`, а иначе данные об ошибке могут быть потеряны.

В следующей программе показано применение `ferror()`. Программа удаляет табуляции из файла, заменяя их соответствующим количеством пробелов. Размер табуляции определяется макросом `TAB_SIZE`. Обратите внимание, что `ferror()` вызывается после каждой операции с файлом. При запуске этой программы указывайте в командной строке имена входного и выходного файлов.

```
/* Программа заменяет в текстовом файле символы табуляции пробелами
   и отслеживает ошибки */
#include <stdio.h>
#include <stdlib.h>

#define TAB_SIZE 8
#define IN 0
#define OUT 1

void err(int e);
int main(int argc, char *argv[])
{
    FILE *in, *out;
    int tab, i;
    char ch;

    if(argc!=3) {
        printf("синтаксис: detab <входной_файл> <выходной_файл>\n");
        exit(1);
    }

    if((in = fopen(argv[1], "rb"))==NULL) {
        printf("Нельзя открыть %s.\n", argv[1]);
        exit(1);
    }

    if((out = fopen(argv[2], "wb")) == NULL) {
        printf("Нельзя открыть %s.\n", argv[2]);
        exit(1);
    }

    tab = 0;
    do {
        ch = getc(in);
        if(ferror(in)) err(IN);

        /* если найдена табуляция, выводится соответствующее число
           пробелов */
        if(ch=='\t') {
            for(i=tab; i<8; i++) {
```

```

        putc(' ', out);
        if(ferror(out)) err(OUT);
    }
    tab = 0;
}
else {
    putc(ch, out);
    if(ferror(out)) err(OUT);
    tab++;
    if(tab==TAB_SIZE) tab = 0;
    if(ch=='\n' || ch=='r') tab = 0;
}
} while(!feof(in));
fclose(in);
fclose(out);

return 0;
}

void err(int e)
{
    if(e==IN) printf("Ошибка при вводе.\n");
    else printf("Ошибка при выводе.\n");
    exit(1);
}

```

Стирание файлов

Функция `remove()` стирает указанный файл. Вот ее прототип:

```
int remove(const char *имя_файла);
```

В случае успешного выполнения эта функция возвращает нуль, а в противном случае — ненулевое значение.

Следующая программа стирает файл, указанный в командной строке. Однако вначале она дает возможность передумать. Утилита, подобная этой, может пригодиться компьютерным пользователям-новичкам.

```

/* Двойная проверка перед стиранием. */
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>

int main(int argc, char *argv[])
{
    char str[80];

    if(argc!=2) {
        printf("синтаксис: xerase <имя_файла>\n");
        exit(1);
    }

    printf("Стереть %s? (Y/N): ", argv[1]);
    gets(str);

    if(toupper(*str)=='Y')
        if(remove(argv[1])) {
            printf("Нельзя стереть файл.\n");
            exit(1);
        }
}

```

```

    }
    return 0;
}

```

Дозапись потока

Для дозаписи содержимого выводного потока в файл применяется функция `fflush()`. Вот ее прототип:

```
int fflush(FILE *уф);
```

Эта функция записывает все данные, находящиеся в буфере в файл, который указан с помощью *уф*. При вызове функции `fflush()` с пустым (`null`) указателем файла *уф* будет выполнена дозапись во все файлы, открытые для вывода.

После своего успешного выполнения `fflush()` возвращает нуль, в противном случае — EOF.

Функции `fread()` и `fwrite()`

Для чтения и записи данных, тип которых может занимать более 1 байта, в файловой системе языка C имеется две функции: `fread()` и `fwrite()`. Эти функции позволяют читать и записывать блоки данных любого типа. Их прототипы следующие:

```
size_t fread(void *буфер, size_t колич_байт, size_t счетчик, FILE *уф);
size_t fwrite(const void *буфер, size_t колич_байт, size_t счетчик, FILE *уф);
```

Для `fread()` *буфер* — это указатель на область памяти, в которую будут прочитаны данные из файла. А для `fwrite()` *буфер* — это указатель на данные, которые будут записаны в файл. Значение *счетчик* определяет, сколько считывается или записывается элементов данных, причем длина каждого элемента в байтах равна *колич_байт*. (Вспомните, что тип `size_t` определяется как одна из разновидностей целого типа без знака.) И, наконец, *уф* — это указатель файла, то есть на уже открытый поток.

Функция `fread()` возвращает количество прочитанных элементов. Если достигнут конец файла или произошла ошибка, то возвращаемое значение может быть меньше, чем *счетчик*. А функция `fwrite()` возвращает количество записанных элементов. Если ошибка не произошла, то возвращаемый результат будет равен значению *счетчик*.

Использование `fread()` и `fwrite()`

Как только файл открыт для работы с двоичными данными, `fread()` и `fwrite()` соответственно могут читать и записывать информацию любого типа. Например, следующая программа записывает в дисковый файл данные типов `double`, `int` и `long`, а затем читает эти данные из того же файла. Обратите внимание, как в этой программе при определении длины каждого типа данных используется функция `sizeof()`.

```

/* Запись несимвольных данных в дисковый файл
   и последующее их чтение. */
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    FILE *fp;
    double d = 12.23;
    int i = 101;

```

```

long l = 123023L;

if((fp=fopen("test", "wb+"))==NULL) {
    printf("Ошибка при открытии файла.\n");
    exit(1);
}

fwrite(&d, sizeof(double), 1, fp);
fwrite(&i, sizeof(int), 1, fp);
fwrite(&l, sizeof(long), 1, fp);

rewind(fp);

fread(&d, sizeof(double), 1, fp);
fread(&i, sizeof(int), 1, fp);
fread(&l, sizeof(long), 1, fp);

printf("%f %d %ld", d, i, l);

fclose(fp);

return 0;
}

```

Как видно из этой программы, в качестве буфера можно использовать (и часто именно так и делают) просто память, в которой размещена переменная. В этой простой программе значения, возвращаемые функциями `fread()` и `fwrite()`, игнорируются. Однако на практике эти значения необходимо проверять, чтобы обнаружить ошибки.

Одним из самых полезных применений функций `fread()` и `fwrite()` является чтение и запись данных пользовательских типов, особенно структур. Например, если определена структура

```

struct struct_type {
    float balance;
    char name[80];
} cust;

```

то следующий оператор записывает содержимое `cust` в файл, на который указывает `fp`:

```
fwrite(&cust, sizeof(struct struct_type), 1, fp);
```

Пример со списком рассылки

Чтобы показать, как можно легко записывать большие объемы данных, пользуясь функциями `fread()` и `fwrite()`, мы переделаем программу работы со списком рассылки, с которой впервые встретились в главе 7. Усовершенствованная версия сможет сохранять адреса в файле. Как и раньше, адреса будут храниться в массиве структур следующего типа:

```

struct addr {
    char name[30];
    char street[40];
    char city[20];
    char state[3];
    unsigned long int zip;
} addr_list[MAX];

```

Значение `MAX` определяет максимальное количество адресов, которое может быть в списке.

При выполнении программы поле name каждой структуры инициализируется пустым указателем (NULL). В программе свободной считается та структура, поле name которой содержит строку нулевой длины, т.е. имя адресата представляет собой пустую строку.

Далее приведены функции save() и load(), которые используются соответственно для сохранения и загрузки базы данных (списка рассылки). Обратите внимание, насколько кратко удалось закодировать каждую из функций, а ведь эта краткость достигнута благодаря мощи fread() и fwrite()! И еще обратите внимание на то, как эти функции проверяют значения, возвращаемые функциями fread() и fwrite(), чтобы обнаружить таким образом возможные ошибки.

```
/* Сохранение списка. */
void save(void)
{
    FILE *fp;
    register int i;

    if((fp=fopen("maillist", "wb"))==NULL) {
        printf("Ошибка при открытии файла.\n");
        return;
    }

    for(i=0; i<MAX; i++)
        if(*addr_list[i].name)
            if(fwrite(&addr_list[i],
                sizeof(struct addr), 1, fp)!=1)
                printf("Ошибка при записи файла.\n");

    fclose(fp);
}

/* Загрузка файла. */
void load(void)
{
    FILE *fp;
    register int i;

    if((fp=fopen("maillist", "rb"))==NULL) {
        printf("Ошибка при открытии файла.\n");
        return;
    }

    init_list();
    for(i=0; i<MAX; i++)
        if(fread(&addr_list[i],
            sizeof(struct addr), 1, fp)!=1) {
            if(feof(fp)) break;
            printf("Ошибка при чтении файла.\n");
        }

    fclose(fp);
}
```

Обе функции, save() и load(), подтверждают (или не подтверждают) успешность выполнения функциями fread() и fwrite() операций с файлом, проверяя значения, возвращаемые функциями fread() и fwrite(). Кроме того, функция load() явно проверяет, не достигнут ли конец файла. Делает она это с помощью вызова функции feof(). Это приходится делать потому, что fread() и в случае ошибки, и при достижении конца файла возвращает одно и то же значение.

Далее показана вся программа, обрабатывающая списки рассылки. Ее можно использовать как ядро для дальнейших расширений, в нее, например, можно добавить средства поиска адресов.

```
/* Простая программа обработки списка рассылки, в которой
   используется массив структур */
#include <stdio.h>
#include <stdlib.h>

#define MAX 100

struct addr {
    char name[30];
    char street[40];
    char city[20];
    char state[3];
    unsigned long int zip;
} addr_list[MAX];

void init_list(void), enter(void);
void delete(void), list(void);
void load(void), save(void);
int menu_select(void), find_free(void);

int main(void)
{
    char choice;

    init_list(); /* инициализация массива структур */
    for(;;) {
        choice = menu_select();
        switch(choice) {
            case 1: enter();
                    break;
            case 2: delete();
                    break;
            case 3: list();
                    break;
            case 4: save();
                    break;
            case 5: load();
                    break;
            case 6: exit(0);
        }
    }

    return 0;
}

/* Инициализация списка. */
void init_list(void)
{
    register int t;

    for(t=0; t<MAX; ++t) addr_list[t].name[0] = '\0';
}

/* Получение значения, выбранного в меню */
int menu_select(void)
```

```

{
    char s[80];
    int c;

    printf("1. Ввести имя\n");
    printf("2. Удалить имя\n");
    printf("3. Вывести список\n");
    printf("4. Сохранить файл\n");
    printf("5. Загрузить файл\n");
    printf("6. Выход\n");

    do {
        printf("\nВведите номер нужного пункта: ");
        gets(s);
        c = atoi(s);
    } while(c<0 || c>6);
    return c;
}

/* Добавление адреса в список. */
void enter(void)
{
    int slot;
    char s[80];

    slot = find_free();

    if(slot==-1) {
        printf("\nСписок заполнен");
        return;
    }

    printf("Введите имя: ");
    gets(addr_list[slot].name);

    printf("Введите улицу: ");
    gets(addr_list[slot].street);

    printf("Введите город: ");
    gets(addr_list[slot].city);

    printf("Введите штат: ");
    gets(addr_list[slot].state);

    printf("Введите почтовый индекс: ");
    gets(s);
    addr_list[slot].zip = strtoul(s, '\0', 10);
}

/* Поиск свободной структуры. */
int find_free(void)
{
    register int t;

    for(t=0; addr_list[t].name[0] && t<MAX; ++t) ;

    if(t==MAX) return -1; /* свободных структур нет */
    return t;
}

```

```

/* Удаление адреса. */
void delete(void)
{
    register int slot;
    char s[80];

    printf("Введите № записи: ");
    gets(s);
    slot = atoi(s);

    if(slot>=0 && slot < MAX)
        addr_list[slot].name[0] = '\0';
}

/* Вывод списка на экран. */
void list(void)
{
    register int t;

    for(t=0; t<MAX; ++t) {
        if(addr_list[t].name[0]) {
            printf("%s\n", addr_list[t].name);
            printf("%s\n", addr_list[t].street);
            printf("%s\n", addr_list[t].city);
            printf("%s\n", addr_list[t].state);
            printf("%lu\n\n", addr_list[t].zip);
        }
    }
    printf("\n\n");
}

/* Сохранение списка. */
void save(void)
{
    FILE *fp;
    register int i;

    if((fp=fopen("maillist", "wb"))==NULL) {
        printf("Ошибка при открытии файла.\n");
        return;
    }

    for(i=0; i<MAX; i++)
        if(*addr_list[i].name)
            if(fwrite(&addr_list[i],
                sizeof(struct_addr), 1, fp)!=1)
                printf("Ошибка при записи файла.\n");

    fclose(fp);
}

/* Загрузить файл. */
void load(void)
{
    FILE *fp;
    register int i;

    if((fp=fopen("maillist", "rb"))==NULL) {

```

```

    printf("Ошибка при открытии файла.\n");
    return;
}

init_list();
for(i=0; i<MAX; i++)
    if(fread(&addr_list[i],
        sizeof(struct addr), 1, fp)!=1) {
        if(feof(fp)) break;
        printf("Ошибка при чтении файла.\n");
    }

fclose(fp);
}

```

Ввод/вывод при прямом доступе: функция fseek()

При прямом доступе можно выполнять операции ввода/вывода, используя систему ввода/вывода языка С и функцию `fseek()`, которая устанавливает указатель текущей позиции в файле. Вот прототип этой функции:

```
int fseek(FILE *уф, long int колич_байт, int начало_отсчета);
```

Здесь *уф* — это указатель файла, возвращаемый в результате вызова функции `fopen()`, *колич_байт* — количество байтов, считая от *начало_отсчета*, оно определяет новое значение указателя текущей позиции, а *начало_отсчета* — это один из следующих макросов:

Начало отсчета	Макрос
Начало файла	SEEK_SET
Текущая позиция	SEEK_CUR
Конец файла	SEEK_END

Поэтому, чтобы получить в файле доступ на расстоянии *колич_байт* байтов от начала файла, *начало_отсчета* должно равняться `SEEK_SET`. Чтобы при доступе расстояние отсчитывалось от текущей позиции, используйте макрос `SEEK_CUR`, а чтобы при доступе расстояние отсчитывалось от конца файла, нужно указывать макрос `SEEK_END`. При успешном завершении своей работы функция `fseek()` возвращает нуль, а в случае ошибки — ненулевое значение.

В следующей программе показано, как используется `fseek()`. Данная программа в определенном файле отыскивает некоторый байт, а затем отображает его. В командной строке нужно указать имя файла, а затем нужный байт, то есть его расстояние в байтах от начала файла.

```

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

int main(int argc, char *argv[])
{
    FILE *fp;

    if(argc!=3) {
        printf("Синтаксис: SEEK <имя_файла> <байт>\n");
        exit(1);
    }

```

```

if((fp = fopen(argv[1], "rb"))==NULL) {
    printf("Ошибка при открытии файла.\n");
    exit(1);
}

if(fseek(fp, atol(argv[2]), SEEK_SET)) {
    printf("Сбой при поиске.\n");
    exit(1);
}

printf("В %ld-м байте содержится %c.", atol(argv[2]), getc(fp));
fclose(fp);

return 0;
}

```

Функцию `fseek()` можно использовать для доступа внутри многих значений одного типа, просто умножая размер данных на номер элемента, который вам нужен. Например, предположим, имеется список рассылки, который состоит из структур типа `addr` (определенных ранее). Чтобы получить доступ к десятому адресу в файле, в котором хранятся адреса, используйте следующий оператор:

```

fseek (fp, 9*sizeof(struct addr), SEEK_SET);

```

Текущее значение указателя текущей позиции в файле можно определить с помощью функции `ftell()`. Вот ее прототип:

```

long int ftell(FILE *uf);

```

Функция возвращает текущее значение указателя текущей позиции в файле, связанном с указателем файла `uf`. При неудачном исходе она возвращает `-1`.

Обычно прямой доступ может потребоваться лишь для двоичных файлов. Причина тут простая — так как в текстовых файлах могут выполняться преобразования символов, то может и не быть прямого соответствия между тем, что находится в файле и тем байтом, к которому нужен доступ. Единственный случай, когда надо использовать `fseek()` для текстового файла — это доступ к той позиции, которая была уже найдена с помощью `ftell()`; такой доступ выполняется с помощью макроса `SEEK_SET`, используемого в качестве начала отсчета.

Хорошо помните следующее: даже если в файле находится один только текст, все равно этот файл при необходимости можно открыть и в двоичном режиме. Никакие ограничения, связанные с тем, что файлы содержат текст, к операциям прямого доступа не относятся. Эти ограничения относятся только к файлам, открытым *в текстовом режиме*.

Функции `fprintf()` и `fscanf()`

Кроме основных функций ввода/вывода, о которых шла речь, в системе ввода/вывода языка C также имеются функции `fprintf()` и `fscanf()`. Эти две функции, за исключением того, что предназначены для работы с файлами, ведут себя точно так же, как и `printf()` и `scanf()`. Прототипы функций `fprintf()` и `fscanf()` следующие:

```

int fprintf(FILE *uf, const char *управляющая_строка,...);
int fscanf(FILE *uf, const char *управляющая_строка,...);

```

где `uf` — указатель файла, возвращаемый в результате вызова `fopen()`. Операции ввода/вывода функции `fprintf()` и `fscanf()` выполняют с тем файлом, на который указывает `uf`.

В качестве примера предлагается рассмотреть следующую программу, которая читает с клавиатуры строку и целое значение, а затем записывает их в файл на диске; имя этого файла — TEST. После этого программа читает этот файл и выводит информацию на экран. После запуска программы проверьте, каким получится файл TEST. Как вы и увидите, в нем будет вполне удобочитаемый текст.

```
/* пример использования fscanf() и fprintf() */
#include <stdio.h>
#include <io.h>
#include <stdlib.h>

int main(void)
{
    FILE *fp;
    char s[80];
    int t;

    if((fp=fopen("test", "w"))==NULL) {
        printf("Ошибка при открытии файла.\n");
        exit(1);
    }

    printf("Введите строку и число: ");
    fscanf(stdin, "%s%d", s, &t); /* читать с клавиатуры */

    fprintf(fp, "%s %d", s, t); /* писать в файл */
    fclose(fp);

    if((fp=fopen("test", "r"))==NULL) {
        printf("Ошибка при открытии файла.\n");
        exit(1);
    }

    fscanf(fp, "%s%d", s, &t); /* чтение из файла */
    fprintf(stdout, "%s %d", s, t); /* вывод на экран */

    return 0;
}
```

Маленькое предупреждение. Хотя читать разноразличные данные из файлов на дисках и писать их в файлы, расположенные также на дисках, часто легче всего именно с помощью функций `fprintf()` и `fscanf()`, но это не всегда самый эффективный способ выполнения операций чтения и записи. Так как данные в формате ASCII записываются так, как они должны появиться на экране (а не в двоичном виде), то каждый вызов этих функций сопряжен с определенными накладными расходами. Поэтому, если надо заботиться о размере файла или скорости, то, скорее всего, придется использовать `fread()` и `fwrite()`.

Стандартные потоки

Что касается файловой системы языка C, то в начале выполнения программы автоматически открываются три потока. Это `stdin` (стандартный поток ввода), `stdout` (стандартный поток вывода) и `stderr` (стандартный поток ошибок). Обычно эти потоки направляются к консоли, но в средах, которые поддерживают перенаправление ввода/вывода, они могут быть перенаправлены операционной системой на другое уст-

ройство. (Перенаправление ввода/вывода поддерживается, например, такими операционными системами, как Windows, DOS, UNIX и OS/2.)

Так как стандартные потоки являются указателями файлов, то они могут использоваться системой ввода/вывода языка C также для выполнения операций ввода/вывода на консоль. Например, `putchar()` может быть определена таким образом:

```
int putchar(char c)
{
    return putc(c, stdout);
}
```

Вообще говоря, `stdin` используется для считывания с консоли, а `stdout` и `stderr` — для записи на консоль.

В роли указателей файлов потоки `stdin`, `stdout` и `stderr` можно применять в любой функции, где используется переменная типа `FILE *`. Например, для ввода строки с консоли можно написать примерно такой вызов `fgets()`:

```
char str[255];
fgets(str, 80, stdin);
```

И действительно, такое применение `fgets()` может оказаться достаточно полезным. Как уже говорилось в этой книге, при использовании `gets()` не исключена возможность, что массив, который используется для приема вводимых пользователем символов, будет переполнен. Это возможно потому, что `gets()` не проводит проверку на отсутствие нарушения границ. Полезной альтернативой `gets()` является функция `fgets()` с аргументом `stdin`, так как эта функция может ограничивать число читаемых символов и таким образом не допустить переполнения массива. Единственная проблема, связанная с `fgets()`, состоит в том, что она не удаляет символ новой строки (в то время как `gets()` удаляет!), поэтому его приходится удалять “вручную”, как показано в следующей программе:

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

int main(void)
{
    char str[80];
    int i;

    printf("Введите строку: ");
    fgets(str, 10, stdin);

    /* удалить символ новой строки, если он есть */
    i = strlen(str)-1;
    if(str[i]=='\n') str[i] = '\0';

    printf("Это ваша строка: %s", str);

    return 0;
}
```

Не забывайте, что `stdin`, `stdout` и `stderr` — это не переменные в обычном смысле, и им нельзя присваивать значение с помощью `fopen()`. Кроме того, именно потому, что в начале работы программы эти указатели файлов создаются автоматически, в конце работы они и закрываются автоматически. Так что и не пытайтесь самостоятельно их закрыть.

Связь с консольным вводом/выводом

В языке С консольный и файловый ввод/вывод не слишком отличаются друг от друга. Функции консольного ввода/вывода, описанные в главе 8, на самом деле направляют результаты своих операций на один из потоков — `stdin` или `stdout`, и по сути, каждая из них является специальной версией соответствующей файловой функции. Функции консольного ввода/вывода для того и существуют, чтобы было удобно именно программисту.

Как говорилось в предыдущем разделе, ввод/вывод на консоль можно выполнять с помощью любой файловой функции языка С. Однако для вас может быть сюрпризом, что, оказывается, операции ввода/вывода на дисковых файлах можно выполнять с помощью функции консольного ввода/вывода, например, `printf()`! Дело в том, что все функции консольного ввода/вывода, о которых говорилось в главе 8, выполняют свои операции с потоками `stdin` и `stdout`. В средах, поддерживающих перенаправление ввода/вывода, это равносильно тому, что `stdin` или `stdout` могут быть перенаправлены на устройство, отличное от клавиатуры или экрана. Проанализируйте, например, следующую программу:

```
#include <stdio.h>

int main(void)
{
    char str[80];

    printf("Введите строку: ");
    gets(str);
    printf(str);

    return 0;
}
```

Предположим, что эта программа называется `TEST`. При ее нормальном выполнении на экран выводится подсказка, затем читается строка, введенная с клавиатуры, и, наконец, эта строка выводится на экран. Однако в средах, в которых поддерживается перенаправление ввода/вывода, один из потоков `stdin` или `stdout` (или оба одновременно) можно перенаправить в файл. Например, в среде DOS или Windows следующий запуск `TEST`

```
TEST > OUTPUT
```

приводит к тому, что вывод этой программы будет записан в файл по имени `OUTPUT`. А следующий запуск `TEST`

```
TEST < INPUT > OUTPUT
```

направляет поток `stdin` в файл по имени `INPUT`, а поток стандартного вывода — в файл по имени `OUTPUT`.

Когда С-программа завершается, то все перенаправленные потоки возвращаются в состояния, которые были установлены по умолчанию.

Перенаправление стандартных потоков: функция `freopen()`

Для перенаправления стандартных потоков можно воспользоваться функцией `freopen()`. Эта функция связывает имеющийся поток с новым файлом. Так что она вполне может связать с новым файлом и стандартный поток. Вот прототип этой функции:

```
FILE *freopen(const char *имя_файла, const char *режим, FILE *поток);
```


где *имя_файла* — это указатель на имя файла, который требуется связать с потоком, на который указывает указатель *поток*. Файл открывается в режиме *режим*; этот параметр может принимать те же значения, что и соответствующий параметр функции `fopen()`. Если функция `freopen()` выполнилась успешно, то она возвращает *поток*, а если встретились ошибки, — то `NULL`.

В следующей программе показано использование функции `freopen()` для перенаправления стандартного потока вывода `stdout` в файл с именем `OUTPUT`.

```
#include <stdio.h>

int main(void)
{
    char str[80];

    freopen("OUTPUT", "w", stdout);

    printf("Введите строку: ");
    gets(str);
    printf(str);

    return 0;
}
```

Вообще говоря, перенаправление стандартных потоков с помощью `freopen()` в некоторых случаях может быть полезно, например, при отладке. Однако выполнение дисковых операций ввода/вывода на перенаправленных потоках `stdin` и `stdout` не настолько эффективно, как использование таких функций, как `fread()` или `fwrite()`.

Полный
справочник по



Глава 10

Препроцессор и комментарии

В исходный код программы на языке С можно вставлять различные инструкции компилятору. Они называются *директивами препроцессора* и расширяют возможности среды программирования. Кроме них, в этой главе еще рассказывается о комментариях.

Препроцессор

Имеются следующие директивы препроцессора:

#define	#endif	#ifdef	#line
#elif	#error	#ifndef	#pragma
#else	#if	#include	#undef

Как легко заметить, все они начинаются со знака #. Кроме того, каждая директива препроцессора должна занимать отдельную строку. Например, строка

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

рассматривается как недопустимая.



Директива #define

Директива #define определяет идентификатор и последовательность символов, которая будет подставляться вместо идентификатора каждый раз, когда он встретится в исходном файле. Идентификатор называется *именем макроса*, а сам процесс замены — *макрозаменой*¹. В общем виде директива выглядит таким образом:


```
#define имя_макроса последовательность_символов
```

Обратите внимание, что в этом выражении нет точки с запятой. Между идентификатором и последовательностью символов *последовательность_символов* может быть любое количество пробелов, но признаком конца последовательности символов может быть только разделитель строк.




Предположим, например, что вместо значения 1 нужно использовать слово LEFT (левый), а вместо значения 0 — слово RIGHT (правый). Тогда можно сделать следующие объявления с помощью директивы #define:

```
 #define LEFT 1
 #define RIGHT 0
```

В результате компилятор будет подставлять 1 или 0 каждый раз, когда в вашем файле исходного кода встречается идентификатор соответственно LEFT или RIGHT. Например, следующий код выводит на экран 0 1 2:

```
 printf("%d %d %d", RIGHT, LEFT, LEFT+1);
```

После определения имя макроса можно использовать в определениях других имен макросов. Вот, например, код, определяющий значения ONE (один), TWO (два) и THREE (три):

```
 #define ONE 1
 #define TWO ONE+ONE
 #define THREE ONE+TWO
```

¹ А также *макросрасширением*, *макрогенерацией* и *макроподстановкой*. Определение макроса часто называют *макроопределением*, а обращение к макросу — *макросызовом* или *макрокомандой*. Впрочем, иногда макроопределение также называется макрокомандой. — *Прим. ред.*

Макроподстановка — это просто замена какого-либо идентификатора связанной с ним последовательностью символов. Поэтому если требуется определить стандартное сообщение об ошибке, то можно написать примерно следующее:

```
#define E_MS "стандартная ошибка при вводе\n"
/* ... */
printf(E_MS);
```

Теперь каждый раз, когда встретится идентификатор `E_MS`, компилятор будет его заменять строкой “стандартная ошибка при вводе\n”. Для компилятора выражение `printf()` на самом деле будет выглядеть таким образом:

```
printf("стандартная ошибка при вводе\n");
```

Если идентификатор находится внутри строки, заключенной в кавычки, то замены не будет. Например, при выполнении кода

```
#define XYZ это проверка
printf("XYZ");
```

вместо сообщения это проверка будет выводиться последовательность символов `XYZ`.

Если *последовательность символов* не помещается в одной строке, то эту последовательность можно продолжить на следующей строке, поместив в конце предыдущей, как показано ниже, обратную косую черту:

```
#define LONG_STRING "это очень длинная \
строка, используемая в качестве примера"
```

Программисты, пишущие программы на языке C, в именах определяемых идентификаторов часто используют буквы верхнего регистра. Если разработчики программ следуют этому правилу, то тот, кто будет читать их программу, с первого взгляда поймет, что будет происходить макрозамена. Кроме того, все директивы `#define` обычно лучше всего помещать в самом начале файла или в отдельном заголовочном файле, а не разбрасывать по всей программе.

Имена макросов часто используются для определения имен так называемых “магических чисел” (встречающихся в программе). Например, имеется программа, в которой определяется массив и несколько процедур, получающих доступ к этому массиву. Вместо того чтобы размер массива “зашивать в код” в виде константы, этот размер можно определить с помощью оператора `#define`, а затем использовать это имя макроса везде, где требуется размер массива. Таким образом, если требуется изменить этот размер, то потребуется изменить только соответствующий оператор `#define`, а затем перекомпилировать программу. Рассмотрим, например, фрагмент программы

```
#define MAX_SIZE 100
/* ... */
float balance[MAX_SIZE];
/* ... */
for(i=0; i< MAX_SIZE; i++) printf("%f", balance[i]);
/* ... */
for(i=0; i< MAX_SIZE; i++) x += balance[i];
```

Размер массива `balance` определяется именем макроса `MAX_SIZE`, и поэтому если этот размер потребуется в будущем изменить, то надо будет изменить только определение `MAX_SIZE`. В результате при перекомпиляции программы все обращения к этому имени макроса, находящиеся после измененного определения, будут автоматически изменены.

Определение макросов с формальными параметрами

У директивы `#define` имеется еще одно большое достоинство: имя макроса может определяться с формальными параметрами. Тогда каждый раз, когда в программе встречается имя макроса, то используемые в его определении формальные параметры

заменяются теми аргументами, которые встретились в программе. Такого рода макросы называются макросами с *формальными параметрами*¹. Например,

```
#include <stdio.h>

#define ABS(a)  (a) < 0 ? -(a) : (a)

int main(void)
{
    printf("модули чисел -1 и 1 равны соответственно %d и %d", ABS(-1),
ABS(1));

    return 0;
}
```

Во время компиляции этой программы вместо формального параметра *a* из определения макроса будут подставляться значения -1 и 1. Скобки, в которых находится *a*, позволяют в любом случае сделать правильную замену. Например, если скобки, стоящие вокруг *a*, удалить, то выражение

■ `ABS(10-20)`

после макрозамены будет преобразовано в

■ `10-20 < 0 ? -10-20 : 10-20`

и может привести к неправильному результату.

Использование вместо настоящих функций макросов с формальными параметрами дает одно существенное преимущество: увеличивается скорость выполнения кода, потому что в таких случаях не надо тратить ресурсы на вызов функций. Однако если у макроса с формальными параметрами очень большие размеры, то тогда из-за дублирования кода увеличение скорости достигается за счет увеличения размеров программы.

И вот еще что: хотя макросы с формальными параметрами являются полезным средством, но в C99 (и в C++) есть еще более эффективный способ создания машинной программы — с использованием ключевого слова `inline`.

На заметку

В C99 можно определить макрос с переменным количеством формальных параметров; об этом рассказывается в части II этой книги.



Директива `#error`

Директива `#error` заставляет компилятор прекратить компиляцию. Эта директива используется в основном для отладки. В общем виде директива `#error` выглядит таким образом:

`#error сообщение-об-ошибке`

`сообщение-об-ошибке` в двойные кавычки не заключается. Когда встречается директива `#error`, то выводится сообщение об ошибке — возможно, вместе с другой информацией, определяемой компилятором.

¹ А также макроопределениями с параметрами и макросами, напоминающими функции. — Прим. ред.

Директива #include

Директива `#include` дает указание компилятору читать еще один исходный файл — в дополнение к тому файлу, в котором находится сама эта директива. Имя исходного файла должно быть заключено в двойные кавычки или в угловые скобки. Например, обе директивы

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

дают компилятору указание читать и компилировать заголовок для библиотечных функций системы ввода/вывода.

Файлы, имена которых находятся в директивах `#include`, могут в свою очередь содержать другие директивы `#include`. Они называются *вложенными директивами* `#include`. Количество допустимых уровней вложенности у разных компиляторов может быть разным. Однако в стандарте C89 предусмотрено, что компиляторы должны допускать не менее 8 таких уровней. А в стандарте C99 предусмотрена поддержка не менее 15 уровней вложенности.

Способ поиска файла зависит от того, заключено ли его имя в двойные кавычки или же в угловые скобки. Если имя заключено в угловые скобки, то поиск файла проводится тем способом, который определен в компиляторе. Часто это означает поиск определенного каталога, специально предназначенного для хранения таких файлов. Если имя заключено в кавычки, то поиск файла проводится другим способом. Во многих компиляторах это означает поиск файла в текущем рабочем каталоге. Если же файл не найден, то поиск повторяется уже так, как будто имя файла заключено в угловые скобки.

Обычно большинство программистов имена стандартных заголовочных файлов заключают в угловые скобки. А использование кавычек обычно приберегается для имен специальных файлов, относящихся к конкретной программе. Впрочем, твердого и простого правила, по которому кавычки требуется использовать именно таким образом, не существует.

В C-программе директиву `#include` можно использовать не только для указания имени файла, содержащего обычный исходный текст программы, но и для указания *заголовка*. В языке C определен набор стандартных заголовков, содержащих необходимую информацию о различных библиотеках этого языка. Заголовок — это стандартный идентификатор, который может соответствовать имени файла, а может и не соответствовать ему. Таким образом, заголовок является просто абстракцией, которая гарантирует наличие некоторой информации. Однако на практике в языке C заголовки почти всегда являются именами файлов.

Директивы условной компиляции

Имеется несколько директив, которые дают возможность выборочно компилировать части исходного кода вашей программы. Этот процесс называется *условной компиляцией* и широко используется фирмами, живущими за счет коммерческого программного обеспечения — теми, которые поставляют и поддерживают многие специальные версии одной программы.

Директивы `#if`, `#else`, `#elif` и `#endif`

Возможно, самыми распространенными директивами условной компиляции являются `#if`, `#else`, `#elif` и `#endif`. Они дают возможность в зависимости от значения константного выражения включать или исключать те или иные части кода.

В общем виде директива `#if` выглядит таким образом:

```
#if константное_выражение
    последовательность операторов
#endif
```

Если находящееся за `#if` константное выражение истинно, то компилируется код, который находится между этим выражением и `#endif`. В противном случае этот промежуточный код пропускается. Директива `#endif` обозначает конец блока `#if`. Например,

```
/* Простой пример #if. */
#include <stdio.h>

#define MAX 100

int main(void)
{
    #if MAX>99
        printf("Компилируется для массива, размер которого больше 99.\n");
    #endif

    return 0;
}
```

Эта программа выводит сообщение на экран, потому что `MAX` больше 99. В этом примере показано нечто очень важное. Значение выражения, находящегося за директивой `#if`, должно быть вычислено во время компиляции. Поэтому в этом выражении могут находиться только ранее определенные идентификаторы и константы, — но не переменные.

Директива `#else` работает в основном так, как `else` — ключевое слово языка C: задает альтернативу на тот случай, если не выполнено условие `#if`. Предыдущий пример можно дополнить следующим образом:

```
/* Простой пример #if/#else. */
#include <stdio.h>

#define MAX 10

int main(void)
{
    #if MAX>99
        printf("Компилируется для массива, размер которого больше 99.\n");
    #else
        printf("Компилируется для небольшого массива.\n");
    #endif

    return 0;
}
```

В этом случае выясняется, что `MAX` меньше 99, поэтому часть кода, относящаяся к `#if`, не компилируется. Однако компилируется альтернативный код, относящийся к `#else`, и откомпилированная программа будет отображать сообщение Компилируется для небольшого массива.

Обратите внимание, что директива `#else` используется для того, чтобы обозначить и конец блока `#if`, и начало блока `#else`. Это естественно, поскольку любой директиве `#if` может соответствовать только одна директива `#endif`.

Директива `#elif` означает “else if” и устанавливает для множества вариантов компиляции цепочку `if-else-if`. После `#elif` находится константное выражение. Если это выражение истинно, то компилируется находящийся за ним блок кода, и больше не проверяются никакие другие выражения `#elif`. В противном же случае проверяется следующий блок этой последовательности. В общем виде `#elif` выглядит таким образом:

```

#if выражение
    последовательность операторов
#elif выражение 1
    последовательность операторов
#elif выражение 2
    последовательность операторов
#elif выражение 3
    последовательность операторов
#elif выражение 4

.
.
#elif выражение N
    последовательность операторов
#endif

```

Например, в следующем фрагменте для определения знака денежной единицы используется значение `ACTIVE_COUNTRY` (для какой страны):

```

#define US 0
#define ENGLAND 1
#define FRANCE 2

#define ACTIVE_COUNTRY US

#if ACTIVE_COUNTRY == US
    char currency[] = "dollar";
#elif ACTIVE_COUNTRY == ENGLAND
    char currency[] = "pound";
#else
    char currency[] = "franc";
#endif

```

В соответствии со стандартом C89 у директив `#if` и `#elif` может быть не менее 8 уровней вложенности. А в соответствии со стандартом C99 программистам разрешается использовать не менее 63 уровней вложенности. При вложенности каждая директива `#endif`, `#else` или `#elif` относится к ближайшей директиве `#if` или `#elif`. Например, совершенно правильным является следующий фрагмент кода:

```

#if MAX>100
    #if SERIAL_VERSION /* последовательная связь */
        int port=198;
    #elif
        int port=200;
    #endif
#else
    char out_buffer[100];
#endif

```

Директивы `#ifdef` и `#ifndef`

Другой способ условной компиляции — это использование директив `#ifdef` и `#ifndef`, которые соответственно означают “if defined” (если определено) и “if not defined” (если не определено). В общем виде `#ifdef` выглядит таким образом:

```

#ifdef имя_макроса
    последовательность операторов
#endif

```


Блок кода будет компилироваться, если *имя макроса* было определено ранее в операторе `#define`.

В общем виде оператор `#ifndef` выглядит таким образом:

```
#ifndef имя_макроса
    последовательность операторов
#endif
```

Блок кода будет компилироваться, если *имя макроса* еще не определено в операторе `#define`.

И в `#ifdef`, и в `#ifndef` можно использовать оператор `#else` или `#elif`.

Например,

```
#include <stdio.h>

#define TED 10

int main(void)
{
    #ifdef TED
        printf("Привет, Тед.\n");
    #else
        printf("Привет, кто-нибудь.\n");
    #endif
    #ifndef RALPH
        printf("A RALPH не определен, так что Ральфу не повезло:.\n");
    #endif

    return 0;
}
```

выведет Привет, Тед, а также A RALPH не определен, так что Ральфу не повезло.

В соответствии со стандартом C89 допускается не менее 8 уровней `#ifdef` и `#ifndef`. А стандарт C99 устанавливает, что должно поддерживаться не менее 63 уровней вложенности.



Директива `#undef`

Директива `#undef` удаляет ранее заданное определение имени макроса, то есть “аннулирует” его определение; само имя макроса должно находиться после директивы. В общем виде директива `#undef` выглядит таким образом:

```
#undef имя_макроса
```

Вот как, например, можно использовать эту директиву:

```
#define LEN 100
#define WIDTH 100

char array[LEN][WIDTH];

#undef LEN
#undef WIDTH
/* а здесь и LEN и WIDTH уже не определены */
```

И `LEN`, и `WIDTH` определены, пока не встретился оператор `#undef`.

Директива `#undef` используется в основном для того, чтобы локализовать имена макросов в тех участках кода, где они нужны.



Использование `defined`

Кроме применения `#ifdef`, есть еще второй способ узнать, определено ли имя макроса. Можно использовать директиву `#if` в сочетании с оператором времени компиляции `defined`. В общем виде оператор `defined` выглядит таким образом:

```
defined имя_макроса
```

Если *имя_макроса* определено, то выражение считается истинным; в противном случае — ложным. Например, чтобы узнать, определено ли имя макроса `MYFILE`, можно использовать одну из двух команд препроцессора:

```
#if defined MYFILE
```

или

```
#ifdef MYFILE
```

Можно также задать противоположное условие, поставив `!` прямо перед `defined`. Например, следующий фрагмент компилируется только тогда, когда имя макроса `DEBUG` не определено:

```
#if !defined DEBUG
    printf("Окончательная версия!\n");
#endif
```

Единственная причина, по которой используется оператор `defined`, состоит в том, что с его помощью в `#elif` можно узнать, определено ли имя макроса.



Директива `#line`

Директива `#line` изменяет содержимое `__LINE__` и `__FILE__`, которые являются зарезервированными идентификаторами в компиляторе. В первом из них содержится номер компилируемой в данный момент строки кода. А второй идентификатор — это строка, содержащая имя компилируемого исходного файла. В общем виде директива `#line` выглядит таким образом:

```
#line номер "имя_файла"
```

где *номер* — это положительное целое число, которое становится новым значением `__LINE__`, а необязательное *имя_файла* — это любой допустимый идентификатор файла, становящийся новым значением `__FILE__`. Директива `#line` в основном используется для отладки и специальных применений.

Например, следующий код определяет, что счетчик строк будет начинаться с 100, а оператор `printf()` выводит номер 102, потому что он расположен в третьей строке программы после оператора `#line 100`:

```
#include <stdio.h>

#line 100                                /* установить счетчик строк */
int main(void)                          /* строка 100 */
{
    printf("%d\n", __LINE__);           /* строка 101 */
    return 0;
}
```

Директива #pragma

Директива #pragma — это определяемая реализацией директива, которая позволяет передавать компилятору различные инструкции. Например, компилятор может поддерживать трассировку выполнения программы. Тогда возможность трассировки можно указывать в операторе #pragma. Возможности этой директивы и относящиеся к ней подробности должны быть описаны в документации по компилятору.

На заметку

В стандарте C99 директиве #pragma есть альтернатива — оператор _Pragma. О нем рассказывается в части II этой книги.

Операторы препроцессора # и

Имеется два оператора препроцессора: # и ##. Они применяются в сочетании с оператором #define.

Оператор #, который обычно называют оператором *превращения в строку* (stringize), превращает аргумент, перед которым стоит, в строку, заключенную в кавычки. Рассмотрим, например, следующую программу:

```
#include <stdio.h>

#define mkstr(s) # s

int main(void)
{
    printf(mkstr(Мне нравится С.));
    return 0;
}
```

Препроцессор превращает строку

```
printf(mkstr(Мне нравится С.));
```

в

```
printf("Мне нравится С.");
```

Оператор ##, который называют оператором *склеивания* (pasting), или *конкатенации* конкатенирует две лексемы. Рассмотрим, например, программу

```
#include <stdio.h>

#define concat(a, b) a ## b

int main(void)
{
    int xy = 10;
    printf("%d", concat(x, y));
    return 0;
}
```

Препроцессор преобразует

```
printf("%d", concat(x, y));
```

в

```
printf("%d", xy);
```

Если эти операторы покажутся вам незнакомыми, то надо помнить вот о чем: они не являются необходимыми и не используются в большинстве программ. В общем-то, эти операторы предусмотрены для работы препроцессора в некоторых особых случаях.

Имена предопределенных макрокоманд

В языке C определены пять встроенных, предопределенных имен макрокоманд. Вот они:

```
__LINE__
__FILE__
__DATE__
__TIME__
__STDC__
```

В такой же последовательности о них здесь и пойдет речь.

Об именах макросов `__LINE__` и `__FILE__` рассказывалось, когда говорилось о директиве `#line`. Говоря кратко, они содержат соответственно номер строки и имя файла компилируемой программы.

В имени макроса `__DATE__` содержится строка в виде *месяц/день/год*, то есть дата перевода исходного кода в объектный.

В имени макроса `__TIME__` содержится время компиляции программы. Это время представлено строкой, имеющей вид *час:минута:секунда*.

Если `__STDC__` определено как 1, то тогда компилятор выполняет компиляцию в соответствии со стандартом C. А что касается C99, то в этом стандарт определены еще два имени макросов:

```
__STDC_HOSTED__
__STDC_VERSION__
```

`__STDC_HOSTED__` равняется 1 для тех сред, в которых выполнение происходит под управлением операционной системы, и 0 — в противном случае. `__STDC_VERSION__` будет равно как минимум 199901 и будет увеличиваться с каждой новой версией языка C. (В C99 могут быть определены и другие имена макросов, о них рассказывается в части II.)

Комментарии

В стандарте C89 определены комментарии только одного вида; такой комментарий начинается с символов `/*` и заканчивается символами `*/`. Между звездочкой и слешем не должно быть никаких пробелов. Любой текст, расположенный между начальными и конечными символами комментария, компилятором игнорируется. Например, следующая программа только выведет на экран Привет:

```
#include <stdio.h>

int main(void)
{
    printf("Привет");
    /* printf("всем"); */

    return 0;
}
```

Комментарий такого вида называется *многострочным комментарием (multiline comment)*, потому что его текст может располагаться в нескольких строках. Например,

```
/* это  
многострочный  
комментарий */
```

Комментарии могут находиться в любом месте программы, за исключением середины ключевого слова или идентификатора. Приведенный ниже комментарий правильный:

```
x = 10+ /* прибавлять числа */5;
```

а комментарий

```
swi/*такое работать не будет*/tch(c) { ...
```

не является допустимым, потому что комментарий не может разрывать ключевое слово. Впрочем, комментарии обычно не следует размещать и в середине выражений, потому что так труднее разобраться и с выражениями, и с самими комментариями.

Многострочные комментарии не могут быть вложенными. То есть в одном комментарии не может находиться другой. Например, при компиляции следующего фрагмента кода будет обнаружена ошибка:

```
/* это внешний комментарий  
x = y/a;  
/* а это внутренний комментарий, обнаружив который, компилятор  
выдаст сообщение об ошибке */  
*/
```

Однострочные комментарии

В C99 (да и в C++) поддерживается два вида комментариев. Первым из них является `/* */`, или многострочный комментарий, о котором только что говорилось. А вторым — однострочный комментарий. Такой комментарий начинается с символов `//` и заканчивается в конце строки. Например:

```
// это однострочный комментарий
```

Однострочные комментарии особенно полезны тогда, когда нужны краткие, не более чем в одну строку пояснения. Хотя версия C89 такие комментарии официально не поддерживает, зато их признает большинство компиляторов C.

Однострочный комментарий может находиться внутри многострочного комментария. Например, следующий комментарий является вполне допустимым:

```
/* это // проверка вложенных комментариев */
```

Комментарии должны находиться там, где требуется объяснить работу кода. Например, в начале всех функций, за исключением самых очевидных, должен быть комментарий, который сообщает, что именно делает функция, как она вызывается и что возвращает.

Полный справочник по



Часть II

Стандарт C99

Как известно, языки программирования непрерывно развиваются, реагируя на изменения в методологии, приложениях, общепринятой практике и используемом оборудовании. Не является исключением в этом отношении и язык C. Его эволюция пошла двумя путями. Первый — это продолжение разработки самого языка C. Второй путь — это язык C++, для которого C послужил отправной точкой. И хотя последние несколько лет внимание специалистов было приковано к C++,

но никогда не ослабевал их интерес к развитию языка C. Например, в ответ на интернационализацию вычислительной среды, в 1995 году в первоначальный Стандарт C89 были введены различные двух- и многобайтовые функции. После завершения согласования поправок в 1995 году началось общее обновление языка. Конечным результатом этого обновления, конечно же, является C99.

При создании стандарта 1999 года были тщательно перепроверены все элементы языка C, проанализированы типичные способы его использования и сделаны попытки предугадать будущие потребности. Как и ожидалось, фоном для всего творческого процесса послужили “взаимоотношения” C и C++. Получившийся в результате Стандарт C99 является доказательством мощи своего первоисточника. Было изменено очень малое число ключевых элементов C. Говоря кратко, изменения заключаются в том, что было тщательно отобрано небольшое количество дополнений к языку, а также было добавлено несколько новых библиотечных функций. Так что язык C все еще остается языком C!

В части I этой книги рассказывалось о тех возможностях C, которые были определены в Стандарте C89. В этой части мы обсудим возможности, которые появились в C99, а также немногочисленные отличия между C99 и C89.

Полный
справочник по



Глава 11

С99

Возможно, самая большая причина для беспокойства, связанного с появлением нового языкового стандарта, — это вопрос о совместимости со своим предшественником. Устареют ли уже написанные программы после выхода новой спецификации? Были ли изменены важные конструкции? Надо ли менять методологию или технологию программирования? Ответы на эти вопросы часто определяют, в какой степени будет принят новый стандарт и, в дальней перспективе, жизнеспособность самого языка. К счастью, создание C99 было управляемым, беспристрастным процессом — благодаря опытным “диспетчерам” этого процесса. Попросту говоря, если вам нравился C таким, каким он был, то понравится и версия C, определяемая Стандартом C99. То, что многие программисты думали о языке C как о самом элегантном в мире языке программирования, не устарело и сейчас!

В этой главе мы изучим изменения в C и дополнения к C, сделанные Стандартом 1999 года. Многие из этих изменений были вскользь упомянуты еще в части I. Здесь же они будут рассмотрены более подробно. Однако не забывайте, что во время написания этой книги компиляторы, которые поддерживали бы многие новые возможности C99, еще не были широко распространены. Возможно, вам придется немного подождать перед тем, как провести “испытательные полеты” с такими новыми восхитительными конструкциями, которыми являются массивы переменной длины, `restricted-qualified` указатели и тип данных `long long`.



Сравнение C99 с C89. Общее впечатление

Отличия между C99 и C89 можно разбить на три общие категории:

- новые возможности, добавленные к C89
- возможности, удаленные из C89
- возможности, которые были изменены или расширены

Многие из отличий между C89 и C99 достаточно незначительны и относятся лишь к нюансам языка. А в этой книге основное внимание уделяется достаточно заметным изменениям — заметным настолько, чтобы влиять на способ написания программ.

Новые возможности

Скорее всего, самыми заметными из новых средств, которых не было в C89, являются те, которые связаны с использованием новых ключевых слов:

```
inline
restrict
_Bool
_Complex
_Imaginary
```

К другим важным новинкам относятся:

- массивы переменной длины
- поддержка арифметических операций с комплексными числами
- тип данных `long long int`
- комментарий `//`
- возможность распределять код и данные
- добавления к препроцессору
- объявления переменных внутри оператора `for`

- составные литералы
- массивы с переменными границами в качестве членов структур
- назначенные инициализаторы
- изменения в семействе функций `printf()` и `scanf()`
- зарезервированный идентификатор `__func__`
- новые библиотеки и заголовки

Большинство возможностей были созданы комитетом по стандартизации, причем многие из этих возможностей созданы с учетом расширений языка, имеющихся в разных реализациях языка C. Впрочем, в некоторых случаях возможности были позаимствованы у C++, как, например, ключевое слово `inline` и комментарии вида `//`. Важно понять, что при создании C99 не были добавлены классы в стиле C++, наследование и функции-члены. В том, что C должен остаться C — в этом комитет по стандартизации был единодушен.

Удаленные средства

Самым заметным “излишеством”, удаленным при создании C99, было правило “неявного `int`”. В C89 во многих случаях, когда не было явного указания типа данных, подразумевался тип `int`. А в C99 такое не допускается. Также удалено неявное объявление функций. В C89, если функция перед использованием не объявлялась, то подразумевалось неявное объявление. А в C99 такое не поддерживается. Если программа должна быть совместима с C99, то из-за двух этих изменений, возможно, придется немного подправить код.

Измененные средства

При создании C99 было сделано несколько изменений имевшихся средств. По большей части, эти изменения расширяют возможности или вносят определенную ясность в их значение. И только в небольшом количестве случаев изменения ограничивают или сужают применение средства. Многие изменения небольшие, однако некоторые из них являются достаточно важными, в том числе:

- уменьшение ограничений транслятора
- новые целые типы
- расширение правил продвижения целых типов
- более строгие правила употребления оператора `return`

Что касается влияния этих изменений на уже написанные программы, то самое значительный эффект имеет изменение правил употребления оператора `return`, из-за чего код, возможно, придется немного подправить.

В оставшейся части этой главы мы изучим основные различия между C89 и C99.

Указатели, определенные с квалификаторами типа `restrict`

Одной из самых важных новинок, введенных Стандартом C99, является квалификатор типа `restrict` (ограниченный). Этот квалификатор применяется только к указателям. Указатель, определенный с квалификатором типа `restrict`¹, изначально яв-

¹ Указатель, определенный с квалификатором типа `restrict`, называется также *restrict-квалифицированным указателем* или *указателем, квалифицированным как `restrict`*. — *Прим. ред.*

ляется единственным средством, с помощью которого можно получить доступ к указываемому объекту. Доступ к объекту с помощью другого указателя возможен лишь тогда, когда этот второй указатель основан на первом. Таким образом, доступ к объекту возможен только для выражений, составленных на основе указателя с квалификатором типа `restrict`. Такие указатели в основном используются как параметры функций или для указания памяти, распределенной с помощью `malloc()`. Квалификатор типа `restrict` семантики программы не меняет.

Если указатель квалифицирован с помощью квалификатора типа `restrict`, то компилятор может лучше оптимизировать некоторые программы, зная, что указатель с квалификатором типа `restrict` является единственным средством доступа к объекту. Например, если функция имеет два параметра в виде указателей с квалификатором типа `restrict`, то компилятор может допустить, что указатели указывают на разные (причем неперекрывающиеся!) объекты. Проанализируем, например, то, что стало классическим примером применения `restrict` — определение функции `memcpy()`. В C89 у нее имеется следующий прототип:

```
void *memcpy(void *cmp1, const void *cmp2, size_t размер);
```

В описании `memcpy()` сказано, что если объекты, на которые указывают `cmp1` и `cmp2`, перекрываются, то поведение этой функции непредсказуемое. Таким образом, `memcpy()` гарантированно будет работать только с неперекрывающимися объектами.

В C99 можно использовать `restrict`, чтобы в прототипе `memcpy()` явно указать то, что в C89 приходится дополнительно объяснять словами. Вот прототип `memcpy()` в C99:

```
void *memcpy(void * restrict cmp1, const void * restrict cmp2, size_t размер);
```

Квалифицируя `cmp1` и `cmp2` с помощью квалификатора типа `restrict`, в прототипе явно утверждается, что они указывают на неперекрывающиеся объекты.

Из-за преимуществ, которые может принести использование квалификатора типа `restrict`, в C99 он был добавлен в прототипы многих библиотечных функций, определенных еще в C89.



Ключевое слово `inline`

При разработке C99 было добавлено ключевое слово `inline`, которое применяется к функциям. Ставя `inline` в начале объявления функции, вы предлагаете компилятору оптимизировать вызовы к этой функции. Обычно это означает, что при компиляции код этой функции будет вставляться на месте вызовов. Однако ключевое слово `inline` является всего лишь запросом к компилятору и может быть проигнорировано. В C99 особо отмечено, что использование `inline` “предполагает, что вызовы функции должны быть максимально быстрыми”. Спецификатор `inline` также поддерживается в языке C++, и синтаксис C99 для этого ключевого слова совместим с C++.

Чтобы создать *встраиваемую функцию*¹, перед ее определением поставьте ключевое слово `inline`. Например, в следующей программе оптимизируются вызовы функции `max()`:

```
#include <stdio.h>

inline int max(int a, int b)
{
    return a > b ? a : b;
}
```

¹ Называется также *подставляемой функцией*. — Прим. ред.

```
int main(void)
{
    int x=5, y=10;

    printf("Наибольшим из чисел %d и %d является %d\n", x, y,
max(x,y));

    return 0;
}
```

При типичной реализации `inline` предшествующая программа эквивалентна следующей:

```
#include <stdio.h>

int main(void)
{
    int x=5, y=10;

    printf("Наибольшим из чисел %d и %d является %d\n", x, y,
        (x>y ? x : y));

    return 0;
}
```

Причина, по которой встраиваемым функциям придается такое большое значение, состоит в том, что они помогают создавать более эффективный код, поддерживая при этом структурированный, функционально-ориентированный подход. Как вы знаете, каждый раз при вызове функции механизм ее вызова и возврата требует значительного количества ресурсов. Обычно при вызове функции ее аргументы заталкиваются в стек, содержимое различных регистров заносится в память, а затем при возврате функции содержимое этих регистров восстанавливается. Беда в том, что на эти операции требуется время. Однако, если код функции подставляется вместо вызова, такие операции уже не нужны. Впрочем, хотя такие подстановки функции и способствуют ускорению выполнения, но они приводят к увеличению размера кода из-за дублирования кода функции. По этой причине лучше всего использовать `inline` только с очень небольшими функциями, т.е. подставлять код только маленьких функций. Кроме того, хорошо было бы применять это ключевое слово только к тем функциям, которые существенно влияют на производительность программы.

Помните: хотя `inline` обычно приводит к подстановке кода функции на месте ее вызова, компилятор может проигнорировать этот запрос или использовать некоторые другие средства оптимизации вызовов функции.



Новые встроенные типы данных

В стандарте C99 появились новые для C встроенные типы данных. Здесь подробно рассказывается о каждом из них.

Bool

Один из новых типов данных, появившихся в C99, — это `_Bool`, в котором можно хранить значения 1 и 0 (истина (`true`) и ложь (`false`)). `_Bool` представляет собой целый тип данных. Как известно многим читателям, в языке C++ определяется ключевое слово `bool`, которое, несмотря на сходство, все же отличается от `_Bool`. Таким образом, в написании этого типа C99 и C++ несовместимы. Кроме того, в C++ определяются встроенные логи-

ческие константы `true` и `false`, а в C99 этого не делается. Однако в C99 имеется заголовок `<stdbool.h>`, в котором определены имена макросов `bool`, `true` и `false`. Таким образом, можно легко создавать код, совместимый с C/C++.

Причина того, что в качестве ключевого слова указывается `_Bool`, а не `bool`, состоит в том, что во многих уже имеющихся C-программах определены их собственные варианты `bool`. Определяя логический тип как `_Bool`, C99 дает возможность не менять уже написанный код. Однако в новые программы лучше всего вставлять `<stdbool.h>`, а затем использовать имя макроса `bool`.

`_Complex` и `_Imaginary`

Стандарт C99 появился вместе с новой для C поддержкой арифметических операций с комплексными числами; эта поддержка включает в себя ключевые слова `_Complex` и `_Imaginary`, дополнительные заголовки и несколько новых библиотечных функций. Однако никаких реализаций не требуется, чтобы реализовать типы мнимых чисел (`imaginary types`), а автономные приложения (которые обходятся без операционной системы) не обязаны поддерживать комплексные типы. Арифметические операции с комплексными числами появились в C99 для упрощения программирования численных методов.

Определены следующие комплексные типы:

```
float_Complex
float_Imaginary
double_Complex
double_Imaginary
long double_Complex
long double_Imaginary
```

Причина того, что в качестве ключевых слов определены `_Complex` и `_Imaginary`, а не `complex` и `imaginary`, состоит в том, что во многих имеющихся C-программах уже определены их собственные типы комплексных данных, использующие имена `complex` и `imaginary`. Определяя ключевые слова `_Complex` и `_Imaginary`, C99 позволяет не менять уже написанный код.

Заголовок `<complex.h>` определяет (кроме всего прочего) макросы `complex` и `imaginary`, которые в результате макроподстановки превращаются в `_Complex` и `_Imaginary`. Таким образом, в новые программы лучше всего вставлять `<complex.h>`, а затем использовать макросы `complex` и `imaginary`.

Типы целых данных `long long`

В стандарте C99 появились новые для C типы данных `long long int` и `unsigned long long int`. Диапазон значений типа данных `long long int` не уже, чем интервал от $-(2^{63}-1)$ до $(2^{63}-1)$. А диапазон значений типа данных `unsigned long long int` обязан содержать интервал от 0 до $2^{64}-1$. Типы `long long` позволяют поддерживать 64-разрядные целые значения с помощью встроенного типа.



Расширение массивов

В Стандарте C99 появились два новых для C и достаточно важных свойства массивов: переменная длина и возможность включать в объявления массивов квалификаторы типа.

Массивы переменной длины

В С89 размерности массивов необходимо объявлять при помощи выражений из целых констант, причем размер массива фиксируется во время компиляции. В силу определенных обстоятельств, в С99 это правило было изменено. В С99 можно объявить массив, размерности которого определяются любыми допустимыми целыми выражениями, в том числе и такими, значения которых становятся известны только во время выполнения. Такой массив называется *массивом переменной длины* (variable-length array, VLA). Однако такими массивами могут быть только локальные массивы (то есть те, у которых область видимости — прототип или блок). Вот пример массива переменной длины:

```
void f(int dim1, int dim2)
{
    int matrix[dim1][dim2]; /* двумерный массив переменной длины */
    /* ... */
}
```

В данном случае размер `matrix` определяется значениями, передаваемыми функции `f()` через переменные `dim1` и `dim2`. Таким образом, в результате каждого вызова `f()` может получиться массив `matrix` с самыми разными измерениями.

Важно понять, что массивы переменной длины за время “своей жизни” не меняют своих размеров. (Иными словами, они не являются динамическими.) На самом деле массив переменной длины создается с другим размером каждый раз, когда встречается его объявление.

Можно указать массив переменной длины неуказанного размера, используя в качестве размера звездочку, `*`.

Появление массивов переменной длины вызвало небольшое изменение в операторе `sizeof`. Вообще говоря, `sizeof` — это оператор, который вычисляется во время компиляции. То есть во время компиляции он обычно превращается в целую константу, значение которой равно размеру типа или объекта. Однако если применяется к массиву переменной длины, то свое значение он получает только во время выполнения. Это изменение было необходимо потому, что размер массива переменной длины нельзя узнать до времени выполнения.

Одной из главных причин появления массивов переменной длины является желание упростить программирование численных методов. Конечно, это средство применяется довольно широко. Но помните — массивы переменной длины не поддерживаются Стандартом С89 (и в языке С++).

Использование квалификаторов типов в объявлении массива

В С99 при объявлении массива в качестве параметра функции, внутри квадратных скобок этого объявления можно указать ключевое слово `static`. Оно сообщает компилятору, что в массиве, на который указывает этот параметр, всегда будет находиться как минимум названное количество элементов. Например:

```
int f(char str[static 80])
{
    // здесь str всегда является указателем на массив из 80 элементов
    //
}
```

Здесь дается гарантия, что `str` будет указывать на начало массива типа `char`, причем в нем будет не менее 80 элементов.

Внутри квадратных скобок также допускаются ключевые слова `restrict`, `volatile` и `const`, но только для параметров функций. Использование `restrict` означает, что указатель изначально является единственным средством доступа к объекту. Применение `const` показывает, что указатель указывает на один и тот же массив (то есть указатель всегда указывает на один и тот же объект). Можно использовать и `volatile` (означает “асинхронно-изменяемый”), хотя и нет смысла это делать.

Однострочные комментарии

Благодаря Стандарту C99, в языке C появились однострочные комментарии. Комментарий такого вида начинается с `//` и доходит до конца строки. Например,

```
// Это комментарий
int i; // это другой комментарий
```

Однострочные комментарии также поддерживаются языком C++. Они удобны тогда, когда нужны краткие замечания, помещающиеся в одной строке. Многие программисты для длинных описаний используют традиционные для языка C многострочные комментарии, оставляя однострочные комментарии только для объяснений, нужных “по ходу дела”.

Распределение кода и объявлений

В соответствии со Стандартом C89 все объявления, находящиеся внутри блока, должны предшествовать первому оператору кода. Но к Стандарту C99 это правило не относится. Рассмотрим, например, программу

```
#include <stdio.h>

int main(void)
{
    int i;

    i = 10;

    int j; // неправильно для C89, допустимо для C99 и C++

    j = i;

    printf("%d %d", i, j);

    return 0;
}
```

Здесь выражение

```
i = 10;
```

находится между двумя объявлениями: переменной `i` и переменной `j`. Стандарт C89 такое не разрешает. Зато это *вполне* допускается в C99 (да и в C++ тоже). Возможность распределять объявления и код довольно широко используется в языке C++. Появление этой возможности в языке C облегчает написание кода, который можно использовать в средах обоих языков.



Изменения препроцессора

Стандарт C99 внес небольшие изменения и в препроцессор.

Переменные списки аргументов

Возможно, самым важным изменением препроцессора является возможность обрабатывать макросы с переменным количеством аргументов. На переменное количество аргументов указывает многоточие (...), находящееся в определении макроса. Встроенный препроцессорный идентификатор `__VA_ARGS__` определяет, куда будут подставляться аргументы. Например, после включения в программу определения

```
#define MyMax(...) max(__VA_ARGS__)
```

выражение

```
MyMax(a, b);
```

преобразуется в

```
max(a, b);
```

До обозначения переменного количества аргументов (...) макрос может иметь другие аргументы. Например, после определения

```
#define compare(compfunc, ...) compfunc(__VA_ARGS__)
```

оператор

```
compare(strcmp, "один", "два");
```

преобразуется в оператор

```
strcmp("один", "два");
```

Как видно из примера, встроенный идентификатор `__VA_ARGS__` заменяется всеми остальными аргументами.

Оператор `_Pragma`

С выходом C99 в языке C появился еще один способ определять прагму в программе: оператор `_Pragma`. В общем виде этот оператор выглядит таким образом:

```
_Pragma("директива")
```

Здесь *директива* — это вызываемая *прагма*¹. Появление оператора `_Pragma` дает прагмам возможность участвовать в макрозамене.

Встроенные прагмы

В версии C99 определены следующие встроенные прагмы:

Прагма

STDC FP_CONTRACT ON/OFF/DEFAULT

Что означает

Во включенном состоянии (ON) выражения с плавающей точкой считаются неделимыми структурами, которые обрабатываются с помощью аппаратуры. Состояние по умолчанию (DEFAULT) определяется реализацией.

¹ Прагма называется также указанием транслятору, псевдокомментарием, директивой транслятора, указанием компилятору, директивой компилятора. — Прим. ред.

STDC FENV_ACCESS ON/OFF/DEFAULT

Сообщает компилятору, что доступна аппаратура для выполнения операций с плавающей точкой. Состояние по умолчанию определяется реализацией.

STDC CX_LIMITED_RANGE ON/OFF/DEFAULT

Во включенном состоянии (ON) сообщает компилятору, что некоторые формулы с составными значениями являются безопасными. Отключенное состояние (OFF) задается по умолчанию.

Подробные сведения об этих прагмах должны быть приведены в документации по компилятору.

Новые встроенные макросы

К макросам, поддерживаемым C89, в C99 добавлены следующие:

__STDC_HOSTED__
__STDC_VERSION__

1, если имеется операционная система не меньше, чем 199901L; представляет версию языка C

__STDC_IEC_559__

1, если поддерживаются арифметические операции с плавающей запятой IEC 60559

__STDC_IEC_559_COMPLEX__

1, если поддерживаются арифметические операции с комплексными числами IEC 60559

__STDC_ISO_10646__

Значение в виде *gggmmL*, которое указывает год и месяц выхода спецификации ISO/IEC 10646, поддерживаемой компилятором

Объявление переменных внутри цикла for

C99 расширяет возможности цикла `for`, разрешая объявление одной или нескольких переменных в части инициализации цикла. Область видимости переменной, объявленной таким способом, ограничена блоком программы, управляемым выражением `for`. То есть переменная, объявленная внутри цикла `for`, будет локализована внутри этого цикла. Эта возможность появилась в языке C потому, что управляющая переменная цикла `for` часто необходима только внутри этого цикла. А так как эта переменная локализована внутри цикла, то удастся избежать ненужных побочных эффектов.

Вот пример, в котором переменная объявляется в части инициализации цикла `for`:

```
#include <stdio.h>

int main(void)
{
    // объявить i внутри for
    for (int i=0; i < 10; i++)
        printf("%d", i);

    return 0;
}
```

Здесь переменная `i` объявляется внутри цикла `for`, а не до начала его работы.

Как уже говорилось, переменная, объявленная внутри цикла `for`, локализуется внутри этого цикла. Проанализируйте следующую программу. Обратите внимание, что переменная `i` объявляется дважды: в начале `main()` и внутри цикла `for`.

```
#include <stdio.h>

int main(void)
{
    int i = -99;

    // объявить i внутри цикла for
    for (int i=0; i < 10; i++)
        printf("%d ", i);

    printf("\n");

    printf("Значение i равно %d", i); // выводит -99

    return 0;
}
```

Эта программа выводит следующее:

```
0 1 2 3 4 5 6 7 8 9
Значение i равно -99
```

Как показывает вывод, как только заканчивается цикл `for`, заканчивается и область видимости переменной `i`, объявленной внутри этого цикла. Таким образом, последнее выражение `printf()` выводит `-99`, то есть значение `i`, объявленное в начале `main()`.

Возможность объявлять управляющие переменные внутри цикла `for`, уже довольно-таки долгое время имеется в языке C++, и теперь такая возможность используется достаточно широко. Есть надежда, что большинство C-программистов будут делать то же самое.

Составные литералы

C99 дает возможность определять *составные литералы*, которые являются выражениями, состоящими из массивов, структур или объединений; эти выражения и обозначают объекты данного типа. Составной литерал создается путем указания имени типа в круглых скобках, за которым следует список инициализации, обязательно заключенный в фигурные скобки. Когда именем типа является массив, то размер указывать нельзя. Создается безымянный объект.

Вот пример составного литерала:

```
double *fp = (double[]) {1.0, 2.0, 3.0};
```

В данном случае создается указатель на `double`, который называется `fp` и указывает на первый элемент массива, состоящего из трех элементов типа `double`.

Составной литерал, созданный в области видимости файла, существует все время жизни программы. А составной литерал, созданный внутри блока, является локальным объектом, который разрушится, как только при выполнении программы произойдет выход из этого блока.

Массивы с переменными границами в качестве членов структур

C99 дает возможность в качестве последнего члена структуры указывать массив без размера. (В структуре перед гибким массивом-членом должен стоять как минимум еще один член.) Он называется *членом-массивом с переменными границами*. Таким образом, структура может иметь в качестве члена массив переменного размера. В размере такой структуры, возвращаемом `sizeof`, память для гибкого массива не учитывается.

Обычно память для структуры с членом-массивом с переменными границами распределяется автоматически, с помощью `malloc()`. Кроме размера структуры, необходимо еще выделить дополнительную память, чтобы разместить член-массив с переменными границами нужного размера. Например, если имеется следующее определение структуры

```
struct mystruct {  
    int a;  
    int b;  
    float fa[]; // массив с переменными границами  
};
```

то при выполнении следующего кода будет выделяться место для массива из 10 элементов:

```
struct mystruct *p;  
p = (struct mystruct *) malloc(sizeof(struct mystruct) + 10 *  
sizeof(float));
```

Так как `sizeof(struct mystruct)` дает значение, в котором не учтен размер памяти для `fa`, то при вызове `malloc()` с помощью выражения

```
10 * sizeof(float)
```

дополнительно выделяется место для размещения массива из 10 элементов типа `float`.

Назначенные инициализаторы

В C99 появилась новая для C возможность, которая будет особенно полезна для программистов, работающих с разреженными массивами. Это *назначенные инициализаторы*. Такие инициализаторы бывают двух видов: одного вида — для массивов, а другого — для структур и объединений. Для массивов используется назначенные инициализаторы такого вида:

`[индекс] = знач`

где *индекс* указывает элемент, инициализируемый с помощью значения *знач* (то есть тот элемент, которому присваивается начальное значение *знач*). Например,

```
int a[10] = { [0] = 100, [3] = 200 };
```

В данном случае инициализируются только элементы с индексами 0 и 3.

Для членов структур или объединений используется назначенные инициализаторы такого вида:

`.имя-члена`

Применение к структуре назначенного инициализатора позволяет легко инициализировать только нужные члены структуры. Например,

```
struct mystruct {
    int a;
    int b;
    int c;
} ob = { .c = 30, .a = 10 };
```

В данном случае член `b` остается неинициализированным.

Кроме того, применение назначенных инициализаторов дает возможность инициализировать структуру, даже не зная порядка расположения ее членов. Это полезно для предопределенных структур, таких как `div_t`, или для структур, определенных некоторыми независимыми производителями.

Новые возможности семейства функций `printf()` и `scanf()`

В C99 для семейства функций `printf()` и `scanf()` предусмотрена новая возможность: они могут манипулировать с типами данных `long long int` и `unsigned long long int`. Модификатором формата для `long long` является `ll`. Например, в следующем фрагменте показано, как выводить значения типа `long long int` и `unsigned long long int`:

```
long long int val;
unsigned long long int u_val;
printf("%lld %llu", val, u_val);
```

Модификатор `ll` можно применять к спецификаторам формата: `d`, `i`, `o`, `u` и `x` — как для `printf()`, так и для `scanf()`.

В C99 добавлен модификатор `hh`, который применяется для указания `char`-аргумента вместе со спецификаторами формата: `d`, `i`, `o`, `u` и `x`.

Оба модификатора, `ll` и `hh`, можно использовать также вместе со спецификатором `n`.

Спецификаторы формата `a` и `A`, которые были добавлены к `printf()`, заставляют выводить значение с плавающей точкой в шестнадцатеричном формате. Формат значения получается следующий:

```
[-]0xh.hhhhp+d
```

Если используется `A`, то `x` и `p` будут выводиться на верхнем регистре. Спецификаторы формата `a` и `A` были также добавлены к `scanf()` и они читают значение с плавающей точкой.

В C99 разрешается при вызове `printf()` к спецификатору `%f` добавлять модификатор `l` (тогда получится `%lf`), но от этого нет никакой пользы¹. В C89 `%lf` для `printf()` не определяется.

Новые библиотеки C99

В C99 добавлены новые библиотеки и заголовки. Вот они:

Заголовок	Назначение
<code><complex.h></code>	Поддерживает арифметические операции с комплексными числами.
<code><fenv.h></code>	Дает доступ к флажкам состояния вычислителя, выполняющего операции с плавающей точкой и другим сигналам этого вычислителя.

¹ Иногда есть: если строка формата используется и для других целей. — Прим. ред.

<inttypes.h>	Определяет стандартный, переносимый набор имен целых типов. Также поддерживает функции, которые работают с целыми значениями наибольшей разрядности.
<iso646.h>	Добавлен в 1995 году Поправкой 1. Определяет имена макросов, соответствующие разным операторам, таким как && и ^.
<stdbool.h>	Поддерживает логические типы данных. Определяет имена макросов bool, true и false, что помогает обеспечивать совместимость с C++.
<stdint.h>	Определяет стандартный, переносимый набор имен целых типов. Этот заголовок входит в состав <inttypes.h>.
<tgmath.h>	Определяет макросы для родового (абстрактного) типа чисел с плавающей точкой.
<wchar.h>	Добавлен в 1995 году Поправкой 1. Поддерживает многобайтовые и двухбайтовые функции.
<wctype.h>	Добавлен в 1995 году Поправкой 1. Поддерживает многобайтные и двухбайтовые функции классификации.

О содержимом этих заголовков и поддерживаемых ими функциях рассказывается в части III.

Зарезервированный идентификатор __func__

В C99 определен идентификатор `__func__`, который указывает (в виде строкового литерала) имя функции, в которой встречается `__func__`. Например,

```
void strUpper(char *str)
{
    static int i = 0;

    i++;
    printf("Функция %s была вызвана %d раз(a).\n", __func__, i);

    while(*str) {
        *str = toupper(*str);
        str++;
    }
}
```

При первом вызове функции `StrUpper()` появится следующий вывод:

 Функция `StrUpper` была вызвана 1 раз(a).

Расширение граничных значений трансляции

Термин “граничные значения трансляции” означает минимальное число разнообразных элементов, которые должен обрабатывать компилятор C. Сюда входит длина идентификаторов, количество уровней вложенности, количество выражений `case` и допустимое количество членов структуры или объединения. В C99 увеличены некоторые из предельных значений для количества этих элементов несмотря на то, что они и так были достаточно щедро определены Стандартом C89. Вот некоторые примеры:

Граничное значение для количества	C89	C99
уровней вложенности блоков	15	127
уровней вложенности условных включений	8	63
значащих символов во внутреннем идентификаторе	31	63

значащих символов во внешнем идентификаторе	6	31
членов структуры или объединения	127	1023
аргументов при вызове функции	31	127

■ Неявный int больше не поддерживается

Несколько лет назад язык C++ отменил правило неявного `int`, а с приходом C99 этому примеру последовал и язык C. В C89 правило неявного `int` гласит, что если явный спецификатор типа отсутствует, то подразумевается тип `int`. Больше всего это правило применялось к возвращаемому типу функций. В прошлом C-программисты часто пропускали `int` при объявлении функций, которые возвращали значение такого типа. Например, в ранние времена языка C функцию `main()` часто писали примерно так:

```
main()
{
    /* ... */
}
```

При таком подходе возвращаемым типом по умолчанию просто считался `int`. В C99 (да и в C++) такого правила присвоения типа по умолчанию больше нет, и `int` приходится указывать явно, что и делается во всех программах, приведенных в этой книге.

А вот другой пример. В прошлом функция, такая как

```
int isEven(int val)
{
    return !(val%2);
}
```

часто писалась примерно так:

```
/* по умолчанию используется целый тип */
isEven(int val)
{
    return !(val%2);
}
```

В первом экземпляре кода возвращаемый тип `int` указывается явно. Во втором — это подразумевается по умолчанию.

Правило неявного `int` применялось не только к возвращаемым значениям функций (хотя здесь оно применялось чаще всего). Например, в C89 и более ранних версиях функцию `isEven()` можно было писать примерно еще и так:

```
isEven(const val)
{
    return !(val%2);
}
```

Здесь параметр `val` также имеет по умолчанию тип `int` — в этом случае `const int`. И опять, это присвоение типа по умолчанию `int` не поддерживается в Стандарте C99.

■ На заметку

В действительности компилятор, совместимый с C99, может принять код, содержащий неявные типы `int`, даже после того, как выдаст предупреждение об ошибке. Так что иногда можно компилировать и старый код. Однако компилятор, совместимый с C99, не обязан принимать такой код.

Удалены неявные объявления функций

Если в C89 встречался вызов функции до явного объявления, то создавалось неявное объявление этой функции. Это неявное объявление имеет такой вид:

```
extern int имя();
```

В C99 неявные объявления функций не поддерживаются.

На заметку

В действительности компилятор, совместимый с C99, может принять код, содержащий неявные объявления функций, даже после того, как выдаст предупреждение об ошибке. Так что можно компилировать и старый код. Однако компилятор, совместимый с C99, не обязан принимать такой код.

Ограничения на return

В C89 в функции, которая имеет возвращаемый тип, отличный от `void` (т.е. предполагается, что такая функция возвращает значение), может встретиться оператор `return` без выражения. Хотя в результате этого теоретически поведение программы было неопределенным, технически в этом не было ничего “незаконного”. Но в C99 в функции, тип которой отличен от `void`, оператор `return` обязан иметь выражение. То есть в C99 внутри функции, которая согласно определению возвращает значение, любой оператор `return` обязан иметь ассоциированное с ним значение, которое и будет возвращено этой функцией. Таким образом, следующая функция является синтаксически допустимой в C89, но недопустима в C99:

```
int f(void)
{
    /* ... */
    return ; // в C99 этот оператор должен возвращать значение
}
```

Расширенные целые типы

C99 в `<stdint.h>` определяет несколько расширенных целых типов. Расширенные типы включают в себя типы с точной разрядностью, минимальной разрядностью, максимальной разрядностью и самый быстрый целый тип. Вот подборка таких типов:

Расширенный тип	Что означает
<code>int16_t</code>	Тип 16-разрядных целых
<code>int_least16_t</code>	Тип целых, содержащий не менее 16 разрядов
<code>int_fast32_t</code>	Самый быстрый тип целых, содержащий не менее 32 разрядов
<code>intmax_t</code>	Тип самых больших целых
<code>uintmax_t</code>	Тип самых больших целых без знака

Расширенные типы облегчают написание переносимого кода. Более подробно они описаны в части III.



Изменения в правилах продвижения целых типов

В C99 расширены правила продвижения целых типов. В C89 значение типа `char`, `short int` или битового поля `int` можно было использовать в выражении вместо `int` или `unsigned int`. Если продвинутое значение помещалось в `int`, то продвижение выполнялось до `int`; в противном же случае первоначальное значение продвигалось до `unsigned int`.

В C99 каждому целому типу присвоен *ранг*. Например, ранг `long long int` выше, чем ранг `int`, который в свою очередь выше, чем ранг `char` и так далее. В выражении любой целый тип, ранг которого ниже, чем ранг `int` или `unsigned int`, может использоваться вместо `int` или `unsigned int`.

Полный справочник по



Часть III

Стандартная библиотека C

В части III книги рассматривается стандартная библиотека языка C. В главе 12 обсуждаются вопросы редактирования связей, использования библиотек и заголовков. В главах 13–20 описаны функции стандартной библиотеки, причем каждая глава посвящена отдельному разделу библиотеки.

В этой книге дается описание стандартных функций, определенных как для C89, так и для C99. В C99 входят все функции, заданные для C89. Поэтому при наличии

компилятора, поддерживающего стандарт C99, можно пользоваться всеми функциями, описанными в данной части. Для компилятора, поддерживающего стандарт C89, функции, определенные только в C99, недоступны. Кроме того, в стандарт C++ входят функции, определенные для C89, но не входят те из них, что определены в C99. Если функция определена только в C99, то об этом будет сказано в ее описании.

При изучении стандартной библиотеки следует помнить, что большинство создателей компиляторов стараются сделать свою библиотеку как можно более полной. Библиотека конкретного компилятора может содержать большое количество дополнительных функций, не рассматриваемых здесь. Например, стандартная библиотека языка C не содержит никаких графических функций, а также функций вывода на экран, потому что они зависят от вычислительной среды. Тем не менее, в большинстве конкретных компиляторов такие функции есть, поэтому всегда полезно просматривать документацию по используемому компилятору.

Полный
справочник по



Глава 12

**Редактирование связей,
использование библиотек
и заголовков**

Работа по написанию компилятора языка С фактически состоит из двух частей. Первая часть — это написание самого компилятора, который преобразует исходный файл в объектный файл. Вторая часть — разработка стандартной библиотеки. Как ни удивительно, но создать компилятор сравнительно просто. Чаще всего большую часть времени и усилий забирает именно работа над библиотечными функциями. Одна из причин этого заключается в том, что многие функции (такие как функции ввода/вывода) должны взаимодействовать с той операционной системой, для которой написан компилятор. Кроме того, стандартная библиотека С содержит большое количество самых разнообразных функций. Действительно, язык С выделяется среди других именно благодаря богатству и гибкости своих возможностей, заложенных в стандартной библиотеке.

В последующих главах будут описаны библиотечные функции С, а в этой главе речь пойдет о фундаментальных концепциях их использования; мы обсудим процесс редактирования связей¹, библиотеки и заголовки.

Редактор связей

*Редактор связей*² выполняет две функции. Во-первых, как можно заключить по его названию, он комбинирует (компонует, редактирует) различные объектные файлы. Вторая его функция — разрешать адреса вызовов и инструкций загрузки, найденных в редактируемых объектных файлах. Чтобы понять принцип работы редактора связей, рассмотрим подробнее процесс раздельной компиляции.

Раздельная компиляция

Раздельная компиляция — это возможность, позволяющая разбить программу на несколько файлов, скомпилировать каждый из этих файлов отдельно, а потом *скомпоновать*³ их, чтобы в конечном итоге создать исполняемый файл. Результатом работы компилятора является объектный файл, а результатом работы редактора связей — *исполняемый файл*⁴. Редактор связей физически связывает файлы, внесенные в список компоновки, в один программный файл и разрешает внешние ссылки. *Внешняя ссылка* создается каждый раз, когда программа из одного файла ссылается на код из другого файла. Это происходит при вызове функции и при ссылке на глобальную переменную. Например, при компоновке двух приведенных ниже файлов, должна быть разрешена ссылка в файле 2 на идентификатор `count`, объявленный в файле 1. Редактор связей сообщает программе из файла 2, где найти `count`.

Файл 1

```
int count;
void display(void);

int main(void)
{
    count = 10;
    display();

    return 0;
}
```

Файл 2

```
#include <stdio.h>
extern int count;

void display(void)
{
    printf("%d", count);
}
```

¹ Называется также *компоновщиком*. — Прим. ред.

² Иногда называется также *компоновщиком*. Впрочем, компоновщиками обычно называют программы, обладающие несколькими меньшими возможностями, чем развитые редакторы связей. — Прим. ред.

³ Этот процесс называется *редактированием связей*. — Прим. ред.

⁴ Называется также *загрузочным модулем*. — Прим. ред.

Аналогично, редактор связей укажет файлу 1, где находится функция `display()`, чтобы можно было ее вызвать.

При генерации объектного кода функции `display()`, компилятор подставляет в него вместо адреса идентификатора `count` “заполнитель”, т.е. ссылку на внешнее имя, потому что он не располагает информацией о том, где находится `count`. Нечто подобное происходит при компиляции `main()`. Адрес функции `display()` не известен, поэтому вместо него используется “заполнитель”, т.е. ссылка на внешнюю программу. При компоновке этих двух файлов содержащиеся в них внешние ссылки заменяются адресами соответствующих элементов. Являются ли эти адреса абсолютными или переместимыми, — зависит от среды¹.

Переместимые коды и абсолютные коды

В результате работы редактора связей для большинства видов вычислительной среды получается *переместимый код*. Так называют объектный код, который может работать в любой свободной области памяти, способной его уместить. В переместимом объектном файле адрес каждой инструкции вызова или загрузки является не фиксированным, а относительным. Таким образом, адреса в переместимом коде отсчитываются от адреса начала программы. При загрузке программы в память для выполнения, загрузчик преобразует относительные адреса в физические адреса, соответствующие адресам ячеек памяти, в которую загружается программа.

В некоторых вычислительных средах, таких как специализированные устройства управления, в которых для всех программ используется одно и то же адресное пространство, редактор связей подставляет в конечный результат своей работы физические адреса. В этом случае он генерирует *абсолютный код*².

Редактирование связей с оверлеями

Хотя в наше время эта возможность применяется редко, следует отметить, что компиляторы С некоторых вычислительных сред в дополнение к обычным компоновщикам предоставляют компоновщики оверлеев. *Компоновщик оверлеев* работает так же, как и обычный, но он также может создавать оверлеи³. *Оверлей* — это фрагмент объектного⁴ кода, который хранится в файле на диске и загружается для работы только по мере необходимости. Место в памяти, которое отводится для загрузки оверлеев, называется *оверлейной областью памяти*. Оверлеи позволяют создавать и запускать программы, которые занимали бы большую область памяти, чем имеющаяся в наличии, потому что в каждый момент времени в памяти находится только та часть программы, которая нужна.

Чтобы понять, как работают оверлеи, представим, что имеется программа, состоящая из семи объектных файлов, которые называются F1, F2, ..., F7. Допустим, имеющейся свободной памяти недостаточно для загрузки программы, скомпонованной обычным образом из всех объектных файлов. Есть возможность скомпоновать только первые пять файлов, а при большем количестве наступит переполнение памяти. Вый-

¹ Редакторы связей обычно располагают широким набором возможностей, и при необходимости пользователь может указать необходимые параметры. — *Прим. ред.*

² Называется также *программой в абсолютных адресах*. — *Прим. ред.*

³ Создание оверлейных программ — стандартная функция для редакторов связей. Кроме того, редакторы связей обычно могут создавать даже загрузочные модули, загружаемые “вразброс”, т.е. в несмежные участки памяти. Все это имеет огромное значение для систем, в которых отсутствует виртуальная память. Но в системах с виртуальной памятью эти возможности часто выглядят как ненужные излишества. — *Прим. ред.*

⁴ Согласно оригиналу. Все же точнее *часть загрузочного модуля*. — *Прим. ред.*

ти из подобной ситуации можно, дав компоновщику команду создать оверлеи из файлов F5, F6 и F7. При каждом вызове функции, которая содержится в одном из этих файлов, *администратор оверлейной загрузки* (программа, предоставляемая компоновщиком или редактором связей) находит необходимый файл и помещает его в оверлейную область памяти, создавая условия для работы программы. Коды, которые получились при компиляции файлов F1 — F4, остаются резидентными. Данная схема проиллюстрирована на рис. 12.1.

Как читатель, возможно, уже догадался, принципиальным преимуществом оверлеев является возможность создавать с их помощью очень большие программы. Основной же их недостаток и причина редкого использования состоит в том, что процесс загрузки занимает определенное время и в значительной мере влияет на быстродействие программы. Поэтому при использовании оверлеев следует взаимосвязанные между собой функции группировать вместе, чтобы свести к минимуму число загрузок оверлеев.

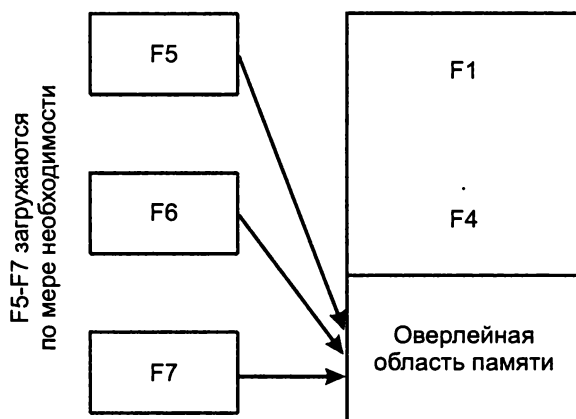


Рис. 12.1 Программа с оверлеями, загруженными в память

Например, если приложение обрабатывает список рассылки, то имеет смысл поместить все подпрограммы сортировки в один оверлей, подпрограммы печати — в другой и т.д.

Как уже было сказано, в современных вычислительных средах оверлеи применяются редко.

Связывание с динамически подсоединяемыми библиотеками (DLL)

Операционная система Windows предоставляет другой вид связывания, так называемое *динамическое связывание*. Динамическое связывание — это процесс, при котором объектный код функции остается в отдельном файле на диске до тех пор, пока не запустится использующая его программа. При запуске такой программы динамически загружаются затребованные, связанные с ней функции. Динамически связанные функции помещаются в специальный тип библиотек, которые называются *динамически подсоединяемыми библиотеками* (DLL — Dynamic-Link Library).

Основным преимуществом таких библиотек является возможность значительно сократить размер исполняемых программ, потому что отпадает необходимость в том, чтобы каждая программа содержала в себе копию используемых библиотечных функций. Другим положительным моментом является то, что при обновлении функций DLL, использующие их программы автоматически используют и все улучшения новых версий.

Стандартная библиотека C не содержится в динамически подключаемой библиотеке, но многие другие типы функций там есть. Например, при написании приложений для Windows, в DLL хранится полный набор функций программного интерфейса приложений (API — Application Program Interface). Нужно отметить, что для программы, написанной на языке C, обычно не имеет значения, хранятся ли библиотечные функции в DLL или в обычном файле библиотек.

Стандартная библиотека C

Содержимое и форма стандартной библиотеки C задается Стандартом ANSI/ISO. Т.е. Стандарт C определяет тот набор функций, который должен поддерживать любой стандартный компилятор. Однако при этом большинство компиляторов предоставляют дополнительные функции, которые не специфицированы в Стандарте. Например, многие компиляторы имеют функции работы с графикой, подпрограммы, управляемые с помощью мышки, и другие им подобные, которых нет в Стандарте C. Пока программа не переносится в другую вычислительную среду, эти нестандартные функции можно использовать без каких-либо негативных последствий. Но если программа должна быть переносимой, применение таких программ нужно ограничить. На самом деле практически все нетривиальные программы C используют нестандартные функции, так что не нужно пугаться и избегать их только из-за того, что они не входят в стандартную библиотеку функций.

Библиотечные файлы и объектные файлы

Хотя библиотеки похожи на объектные файлы, между библиотеками и объектными файлами есть одно важное различие. При компоновке объектных файлов все содержимое каждого объектного файла становится частью конечной исполняемой программы. При этом не важно, используется на самом деле этот код или нет. В случае с файлами библиотек ситуация иная.

Библиотека представляет собой набор функций. В отличие от объектных файлов, в библиотеке каждая функция хранится отдельно. Когда программа использует библиотечную функцию, редактор связей находит эту функцию и добавляет ее код в программу. Таким образом, исполняемый файл содержит только те функции, которые используются программой, а не все библиотечные функции. Поэтому лучше хранить стандартные функции C не в объектных файлах, а в библиотеках, из которых они добавляются в программу избирательно.

Заголовки

С каждой функцией стандартной библиотеки C связан свой заголовок. Соответствующие заголовки используемых функций должны быть включены в программу с помощью директивы `#include`. Заголовки выполняют две важные функции. Во-первых, многие функции стандартной библиотеки работают с данными собственного определенного типа, к которым должна иметь доступ основная программа, использующая эти функции. Эти типы данных задаются в заголовках, связанных с каждой функцией. Одним из наиболее распространенных примеров является заголовок файловой системы `<stdio.h>`, определяющий тип `FILE`, который необходим для выполнения операций с файлами на диске.

Второй причиной включения заголовков является необходимость получения прототипов библиотечных функций. Прототипы функций позволяют компилятору производить

более строгую проверку типов. Хотя прототипы технически являются необязательными, они необходимы для всех практических целей. Кроме того, они нужны для C++. Все программы, содержащиеся в этой книге, подразумевают наличие полного прототипа.

Список стандартных заголовков, определенных Стандартом C89, приведен в таблице 12.1. В таблице 12.2 приведены заголовки, добавленные в Стандарте C99.

Стандартом C для заголовков зарезервированы идентификаторы, начинающиеся символом подчеркивания, за которым следует символ подчеркивания либо заглавная буква.

Как уже было сказано в части I, заголовки — это, как правило, файлы, но не всегда. Компилятор может предопределить содержимое заголовка внутренним образом. Однако в практических целях содержимое стандартных заголовков C находится в файлах, имена которых совпадают с именами самих заголовков.

В следующих главах части III, описывающих все стандартные библиотечные функции, для каждой функции указаны соответствующие ей заголовки.

Таблица 12.1. Заголовки, определенные в C89

Заголовок	Назначение
<assert.h>	Определяет макрос <code>assert()</code>
<ctype.h>	Обработка символов
<errno.h>	Выдача сообщения об ошибках
<float.h>	Задаёт пределы значений с плавающей точкой, зависящие от реализации
<limits.h>	Задаёт различные ограничения, зависящие от реализации
<locale.h>	Поддерживает локализацию
<math.h>	Различные определения, используемые математической библиотекой
<setjmp.h>	Поддерживает нелокальные переходы
<signal.h>	Поддерживает обработку сигналов
<stdarg.h>	Поддерживает списки входных параметров функции с переменным числом аргументов
<stddef.h>	Определяет некоторые наиболее часто используемые константы
<stdio.h>	Поддерживает систему ввода/вывода
<stdlib.h>	Смешанные объявления
<string.h>	Поддерживает функции обработки строк
<time.h>	Поддерживает функции, обращающиеся к системному времени

Таблица 12.2. Заголовки, добавленные в C99

Заголовок	Назначение
<complex.h>	Поддерживает арифметические операции с комплексными числами
<fenv.h>	Предоставляет доступ к флажкам состояния вычислителя, выполняющего операции с плавающей точкой, а также доступ к другим сигналам этого вычислителя
<inttypes.h>	Определяет стандартный, переносимый набор имен целочисленных типов, поддерживает функции, которые работают с целыми значениями наибольшей разрядности
<iso646.h>	Добавлено в 1995 году Поправкой 1; определяет макросы, соответствующие различным операторам, например <code>&&</code> и <code>^</code>
<stdbool.h>	Поддерживает логические типы данных; определяет макрос <code>bool</code> , способствующий совместимости с языком C++
<stdint.h>	Задаёт стандартный переносимый набор имен целочисленных типов; этот файл включен в заголовок <code><inttypes.h></code>
<tgmath.h>	Определяет макросы для родового (абстрактного) типа чисел с плавающей точкой
<wchar.h>	Добавлен в 1995 году Поправкой 1; поддерживает функции обработки многобайтовых слов и двухбайтовых символов
<wctype.h>	Добавлен в 1995 году Поправкой 1; поддерживает функции классификации многобайтовых слов и двухбайтовых символов

Макросы в заголовках

Многие стандартные функции C можно ввести либо как собственно функции, либо как подобные функциям макросы, заданные в заголовке. Например, функцию `abs()`, которая возвращает абсолютную величину целочисленного аргумента, можно также задать как макрос:

```
#define abs(i) (i)<0 ? -(i):(i)
```

Обычно не имеет значения, определена ли стандартная функция как макрос или как обычная функция C. Однако в редких случаях, когда макросы неприменимы, — например, если размер программы должен быть минимальным, или если аргумент нельзя вычислять больше одного раза, — нужно создавать обычные функции и подставлять их вместо макроса. Иногда в самой библиотеке C содержатся функции, которые можно использовать для замены ими макросов.

Чтобы компилятор использовал истинную функцию, необходимо предпринять меры против подстановки им макроса на место имени функции. Для этого есть несколько способов, но, безусловно, лучший из них — просто пометить имя макроса как неопределенное с помощью `#undef`. Например, чтобы заставить компилятор подставить вместо ранее определенного макроса истинную функцию `abs()`, можно в начало программы вставить следующую строку:

```
#undef abs
```

Теперь, когда `abs` больше не является макросом, будет использоваться функция.

Переопределение библиотечных функций

Хотя реализация редакторов связей может иметь некоторые различия, все они в основном работают одинаково. Например, если программа состоит из трех файлов с именами F1, F2 и F3, команда редактора связей выглядит примерно так:

```
LINK F1 F2 F3 LIBC
```

где `LIBC` — это имя стандартной библиотеки.

На заметку

Некоторые редакторы связей используют стандартную библиотеку автоматически и не требуют, чтобы она была задана явно. Кроме того, часто соответствующие библиотечные файлы автоматически включены в интегрированную среду программирования.

С началом процесса редактирования связей редактор связей обычно пытается разрешить все внешние ссылки, ограничившись только файлами F1, F2 и F3. Если это сделано, и еще остались неразрешенные внешние ссылки, редактор связей ищет их в библиотеке.

Пользуясь тем, что большинство редакторов связей работает так, как описано выше, стандартную библиотечную функцию можно переопределить. Например, можно создать свою собственную версию функции `fwrite()`, которая производит обработку выходных данных каким-то определенным образом. В этом случае, при компоновке программы, содержащей определенную программистом версию `fwrite()`, редактор связей находит эту реализацию раньше и использует ее для разрешения всех соответствующих ссылок. Поэтому ко времени просмотра библиотеки неразрешенных ссылок на `fwrite()` не останется, и она не загрузится из библиотеки.

При переопределении библиотечных функций нужно быть очень осторожным, потому что могут возникнуть неожиданные побочные эффекты. Может случиться, что какая-то часть программы использует библиотечную функцию, а эта функция уже переопределена. В этом случае эта часть вместо ожидаемой библиотечной функции получит переопределенную функцию. Например, если функция переопределена для ис-

пользования в одной части программы, а другая часть этой программы использует стандартную библиотечную функцию, то, как минимум, это может привести к непредсказуемому поведению программы. Поэтому лучше просто использовать другие имена для функций, чем переопределять библиотечные функции.

Полный
справочник по



Глава 13

Функции ввода/вывода

В этой главе описаны стандартные функции ввода/вывода в языке C. Сюда вошли функции, определенные как в Стандарте C89, так и в Стандарте C99. С функциями ввода/вывода ассоциирован заголовок `<stdio.h>`. Этот заголовок определяет некоторые макросы и типы, которые используются файловой системой. Наиболее важным из них является тип `FILE`, который используется для объявления указателя на файл. Два других часто используемых типа — `size_t` и `fpos_t`. Тип `size_t`, представляющий собой некоторую разновидность целых без знака, — это тип результата, возвращаемого функцией `sizeof`. Тип `fpos_t` определяет объект, который однозначно задает каждую позицию в файле. Самым популярным макросом, определенным в этом заголовке, является макрос `EOF`, значение которого указывает на конец файла. Другие типы данных и макросы, определенные в заголовке `<stdio.h>`, описаны вместе с функциями, с которыми они связаны.

Многие функции ввода/вывода при возникновении ошибки присваивают встроенной глобальной переменной целого типа `errno` определенное значение. Анализ этой переменной поможет программе получить более подробную информацию о возникшей ошибке. Значения, которые может принимать переменная `errno`, зависят от конкретной реализации компилятора.

В версии C99 введен квалификатор `restrict`, который применяется к некоторым параметрам нескольких функций, первоначально определенных в версии C89. При рассмотрении каждой такой функции будет приведен ее прототип, используемый в среде C89 (который одновременно является прототипом в C++), а параметры с атрибутом `restrict` будут отмечены в описании этой функции.

Обзор системы ввода/вывода приведен в главах 8 и 9 части I.

На заметку

В этой главе описаны функции познакового ввода/вывода. Эти функции были введены в Стандарт C с самого начала и, безусловно, являются наиболее часто используемыми. В 1995 году было добавлено несколько функций, позволяющих обрабатывать символы в расширенном 16-битном алфавите (`wchar_t`); эти функции кратко описаны в главе 19.



Функция `clearerr`

```
#include <stdio.h>
void clearerr(FILE *stream);
```

Функция `clearerr()` сбрасывает (т.е. устанавливает равным нулю) признак ошибки, связанный с потоком, на который указывает элемент `stream`. При этом также сбрасывается признак конца файла.

При успешном обращении к функции `fopen()` признаки ошибок для каждого потока первоначально устанавливаются равными нулю. При работе с файлами ошибки могут возникать по различным причинам, многие из которых зависят от конкретной системы. Истинную природу ошибки можно определить в результате вызова функции `perror()`, которая выводит сообщение, описывающее ошибку (см. описание функции `perror()`).

Пример

Приведенная ниже программа копирует один файл в другой. При возникновении ошибки выводится сообщение, поясняющее ее природу.

```
/* Копирование одного файла в другой. */
#include <stdio.h>
#include <stdlib.h>
```

```

int main (int argc, char *argv [ ] )
{
    FILE *in, *out;
    char ch;
    if (argc!=3) {
        printf("Не введено имя файла.\n");
        exit(1);
    }

    if((in=fopen(argv[1], "rb")) == NULL) {
        printf("Невозможно открыть входной файл. \n");
        exit (1);
    }

    if((out=fopen(argv[2], "wb")) == NULL) {
        printf("Невозможно открыть выходной файл. \n");
        exit(1);
    }

    while(!feof(in)) {
        ch = getc(in);
        if(ferror(in)) {
            printf("Ошибка чтения");
            clearerr(in);
            break;
        } else {
            if(!feof(in)) putc(ch, out);
            if(ferror(out)) {
                printf("Ошибка записи");
                clearerr(out);
                break;
            }
        }
    }
    fclose(in);
    fclose(out);

    return 0;
}

```

Зависимые функции

`feof()`, `ferror()` и `perror()` ...

Функция `fclose`

```

#include <stdio.h>
int fclose(FILE *stream);

```

Функция `fclose()` закрывает файл, связанный с потоком *stream*, и дозаписывает его буфер. После обращения к функции `fclose()` элемент *stream* больше не связан с файлом, и все автоматически выделенные буфера освобождаются.

При успешном выполнении функция `fclose()` возвращает нуль; в противном случае возвращается значение EOF. Попытка закрыть уже закрытый файл расценивается как ошибка. В случае прекращения доступа к носителю данных до закрытия файла, как и при недостатке свободного пространства на диске, будет сгенерирована ошибка.

Пример

Следующая программа открывает и закрывает файл.

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

int main(void)
{
    FILE *fp;

    if((fp=fopen("test", "rb"))==NULL) {
        printf("Не удастся открыть файл.\n");
        exit(1);
    }

    if(fclose(fp))printf("Ошибка при закрытии файла.\n");

    return 0;
}
```

Зависимые функции

fopen(), freopen() и fflush()



Функция feof

```
#include <stdio.h>
int feof(FILE *stream);
```

Функция feof() проверяет, достигнут ли конец файла, связанного с потоком *stream*. Если указатель текущей позиции файла установлен на конец файла, возвращается ненулевое значение; в противном случае возвращается нуль.

При достижении конца файла последующие операции чтения будут возвращать значение EOF до тех пор, пока не будет вызвана функция rewind() или пока указатель текущей позиции файла не будет установлен на новую позицию с помощью функции fseek().

Функция feof() особенно полезна при работе с двоичными файлами, поскольку маркер конца файла также является полноценным двоичным целым. Например, чтобы определить момент достижения конца двоичного файла вместо простой проверки значения, возвращаемого функцией getc(), следует явным образом обратиться к функции feof().

Пример

Данный фрагмент программы показывает один из способов определения конца файла.

```
/*
 * Предполагается, что файл fp был открыт для чтения.
 */
while(!feof(fp)) getc(fp);
```

Зависимые функции

clearerr(), ferror(), perror(), putc() и getc()

Функция `ferror`

```
#include <stdio.h>
int ferror(FILE *stream);
```

Функция `ferror()` проверяет наличие ошибки при работе с файлом, связанным с потоком *stream*. Нулевое значение, возвращаемое этой функцией, говорит о том, что никакой ошибки не обнаружено, а ненулевое значение означает ее наличие.

Чтобы определить природу ошибки, нужно воспользоваться функцией `perror()`.

Пример

Следующий фрагмент программы приводит к аварийному прекращению ее работы при возникновении ошибки.

```
/*
   Предполагается, что fp указывает на поток,
   открытый для записи.
*/

while(!done) {
    putc(info, fp);
    if(ferror(fp)) {
        printf("Ошибка при работе с файлом\n");
        exit(1);
    }
}
```

Зависимые функции

`clearerr()`, `feof()` и `perror()`

Функция `fflush`

```
#include <stdio.h>
int fflush(FILE *stream);
```

Если поток *stream* связан с файлом, открытым для записи, то при обращении к функции `fflush()` в этот файл будет физически записано содержимое выходного буфера. При этом файл остается открытым.

Нулевое значение, возвращаемое функцией, свидетельствует о ее успешном выполнении, а значение EOF — о возникновении ошибки при записи.

При нормальном завершении программы или при заполнении буферов все их содержимое автоматически дозаписывается в файл. Кроме того, буфер дозаписывается в файл при закрытии файла.

Пример

Приведенный фрагмент программы дозаписывает в файл содержимое буфера после каждой операции записи.

```
/*
   Предполагается, что fp связан с выходным файлом
*/
```

```
for(i=0; i<MAX; i++) {
    fwrite(buf, sizeof(some_type), 1, fp);
    fflush(fp);
}
```

Зависимые функции

`fclose()`, `fopen()`, `fread()`, `fwrite()`, `getc()` и `putc()`



Функция `fgetc`

```
#include <stdio.h>
int fgetc(FILE *stream);
```

Функция `fgetc()` возвращает символ, взятый из входного потока *stream* и находящийся сразу после текущей позиции, а также увеличивает указатель текущей позиции файла. Этот символ читается как значение типа `unsigned char`, преобразованное в целое.

При достижении конца файла функция `fgetc()` возвращает значение `EOF`. Но поскольку значение `EOF` является действительным целым значением, при работе с двоичными файлами для обнаружения конца файла необходимо использовать функцию `feof()`. Если функция `fgetc()` обнаруживает ошибку, она возвращает значение `EOF`. Для выявления ошибок, возникающих при работе с двоичными файлами, необходимо использовать функцию `ferror()`.

Пример

Следующая программа читает и выводит на экран содержимое текстового файла.

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

int main(int argc, char *argv[])
{
    FILE *fp;
    char ch;

    if((fp=fopen(argv[1], "r"))==NULL) {
        printf("Невозможно открыть файл.\n");
        exit(1);
    }

    while((ch=fgetc(fp)) != EOF) {
        printf("%c", ch);
    }
    fclose(fp);

    return 0;
}
```

Зависимые функции

`fputc()`, `getc()`, `putc()` и `fopen()`



Функция `fgetpos`

```
#include <stdio.h>
int fgetpos(FILE *stream, fpos_t *position);
```

Функция `fgetpos()` сохраняет в объекте, на который указывает параметр *position*, текущее значение указателя позиции файла из заданного потока. Объект, адресуемый элементом *position*, должен иметь тип `fpos_t`. Сохраняемое значение может быть полезно только для последующего обращения к функции `fsetpos()`.

Отметим, что в версии C99 к параметрам *stream* и *position* применяется квалификатор `restrict`.

При возникновении ошибки функция `fgetpos()` возвращает ненулевое значение; в противном случае возвращается нуль.

Пример

Следующий фрагмент программы присваивает переменной `file_loc` текущее значение положения файла.

```
FILE *fp;
fpos_t file_loc;
.
.
.
fgetpos(fp, &file_loc);
```

Зависимые функции

`fsetpos()`, `fseek()` и `ftell()`



Функция `fgets`

```
#include <stdio.h>
char *fgets(char *str, int num, FILE *stream);
```

Функция `fgets()` читает из входного потока *stream* не более *num-1* символов и помещает их в массив символов, адресуемый указателем *str*. Символы читаются до тех пор, пока не будет прочитан символ новой строки или значение EOF, либо пока не будет достигнут заданный предел. По завершении чтения символов сразу же за последним из них размещается нулевой символ. Символ новой строки сохраняется и становится частью массива, адресуемого элементом *str*.

В версии C99 к параметрам *str* и *stream* применен квалификатор `restrict`.

При успешном выполнении функция `fgets()` возвращает значение *str*, а в случае сбоя — нулевой указатель. В случае ошибки содержимое массива, к которому отсылает указатель *str*, не определено. Поскольку функция `fgets()` возвращает нулевой указатель и при возникновении ошибки, и при достижении конца файла, то для выяснения, что же произошло на самом деле, необходимо использовать функцию `feof()` или `ferror()`.

Пример

Приведенная программа использует функцию `fgets()` для вывода содержимого текстового файла, имя которого задано первым аргументом командной строки.

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

```
int main(int argc, char *argv[])
{
    FILE *fp;
    char str[128];

    if((fp=fopen(argv[1], "r"))==NULL) {
        printf("Не удается открыть файл.\n");
        exit(1);
    }

    while(!feof(fp)) {
        if(fgets(str, 126, fp)) printf("%s", str);
    }
    fclose(fp);

    return 0;
}
```

Зависимые функции

fputs(), fgetc(), gets() и puts()

Функция fopen

```
#include <stdio.h>
FILE *fopen(const char *fname, const char *mode);
```

Функция `fopen()` открывает файл, имя которого задается параметром *fname*, и возвращает указатель на поток, связанный с этим файлом. Типы операций, которые разрешено выполнять с файлом, определяются параметром *mode*. Возможные значения параметра *mode* приведены в таблице 13.1. Строка символов, которая будет играть роль имени реального файла, должна определять его имя, допустимое в данной операционной системе. Эта строка может включать спецификацию пути, если среда поддерживает такую возможность.

В версии C99 к параметрам *fname* и *mode* применен квалификатор `restrict`.

Если функция `fopen()` успешно открыла заданный файл, она возвращает указатель `FILE`. Если файл открыть не удастся, возвращается нулевой указатель.

Таблица 13.1. Допустимые значения параметра *mode* функции `fopen()`

Режим	Назначение
"r"	Открывает текстовый файл для чтения
"w"	Создает текстовый файл для записи
"a"	Дописывает в текстовый файл
"rb"	Открывает двоичный файл для чтения
"wb"	Создает двоичный файл для записи
"ab"	Дописывает в двоичный файл
"r+"	Открывает текстовый файл для чтения и записи
"w+"	Создает текстовый файл для чтения и записи
"a+"	Открывает текстовый файл для чтения и записи
"rb+" или "r+b"	Открывает двоичный файл для чтения и записи
"wb+" или "w+b"	Создает двоичный файл для чтения и записи
"ab+" или "a+b"	Открывает двоичный файл для чтения и записи

Как видно из таблицы, файл можно открывать либо в текстовом, либо в двоичном режиме. В текстовом режиме выполняются преобразования некоторых символов. Например, символы новой строки преобразуются в комбинацию кодов возврата каретки (ASCII 13) и конца строки (ASCII 10). В двоичном режиме подобные преобразования не выполняются.

В следующем фрагменте программы иллюстрируется корректный способ открытия файла.

```
FILE *fp;

if ((fp = fopen("test", "w"))==NULL) {
    printf("Не удается открыть файл.\n");
    exit(1);
}
```

Благодаря такому методу перед записью в файл выявляется любая ошибка, возникающая при его открытии, например, использование защищенного от записи или заполненного диска.

Если с помощью функции `fopen()` открывается файл для вывода (записи), то любой уже существующий файл с заданным именем удаляется, а вместо него создается новый. Если файл с таким именем не существует, он будет создан. Чтобы открыть файл для выполнения операций чтения, нужно, чтобы этот файл уже существовал. В противном случае функция возвратит значение ошибки. Чтобы добавить данные в конец файла, необходимо использовать режим "a". Если окажется, что указанный файл не существует, он будет создан.

Осуществляя доступ к файлу, который открыт для чтения и записи, не следует сразу за операцией ввода выполнять операцию вывода, не прибегнув прежде к промежуточному вызову одной из следующих функций: `fflush()`, `fseek()`, `fsetpos()` или `rewind()`. Нельзя также сразу за операцией вывода выполнять операцию ввода, не прибегнув прежде к промежуточному вызову одной из перечисленных выше функций. Исключением является момент достижения конца файла во время операции ввода, т.е. в конце файла вывод может непосредственно следовать за вводом.

Максимальное количество файлов, которые могут быть открыты одновременно, ограничивается значением `FOPEN_MAX`, определенным в заголовке `<stdio.h>`.

Пример

Следующий фрагмент открывает файл с названием TEST для чтения-записи в двоичном режиме.

```
FILE *fp;

if ((fp = fopen("test", "rb+"))==NULL) {
    printf("Не удается открыть файл.\n");
    exit(1);
}
```

Зависимые функции

`fclose()`, `fread()`, `fwrite()`, `putc()` и `getc()`



Функция `fprintf`

```
#include <stdio.h>
int fprintf(FILE *stream, const char *format,...);
```

Функция `fprintf()` выводит в поток, адресуемый параметром *stream*, значения аргументов, составляющих список аргументов, в соответствии с заданной строкой формата *format*. Возвращаемое значение равно количеству реально выведенных символов. Если при выводе возникла ошибка, возвращается отрицательное число.

В версии C99 к параметрам *stream* и *format* применен квалификатор `restrict`.

Операции преобразования, заданные в строке формата, и команды вывода аналогичны операциям и командам, используемым в функции `printf()`; их полное описание приводится в разделе, посвященном функции `printf`.

Пример

Приведенная программа создает файл с названием TEST и записывает в него строку это тест 10 20.01 в формате, заданном функцией `fprintf()`.

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

int main(void)
{
    FILE *fp;

    if((fp=fopen("test","wb"))==NULL) {
        printf("Не удастся открыть файл.\n");
        exit(1);
    }

    fprintf(fp, "это тест %d %f", 10, 20.01);
    fclose(fp);

    return 0;
}
```

Зависимые функции

`printf()` и `fscanf()`

Функция `fputc`

```
#include <stdio.h>
int fputc(int ch, FILE *stream);
```

Функция `fputc()` записывает символ *ch* в текущую позицию потока *stream*, а затем увеличивает указатель текущей позиции файла. Хотя на практике при объявлении символа *ch* он всегда имеет тип `int`, функцией `fputc()` тип символа преобразуется в `unsigned char`. Поскольку в момент вызова символьный аргумент преобразуется к целому типу, в качестве аргументов обычно можно использовать и символьные переменные. При использовании целого значения, старший байт попросту отбрасывается.

Значением, возвращаемым функцией `fputc()`, является значение записанного символа. При возникновении ошибки возвращается значение EOF. Если файл открыт для выполнения операций в двоичном режиме, значение EOF тоже может оказаться символом. Поэтому, чтобы определить, возникла ли ошибка на самом деле, в таких случаях придется использовать функцию `ferror()`.

Пример

Приведенная функция записывает в заданный поток содержимое строки.

```
void write_string(char *str, FILE *fp)
{
    while(*str) if(!ferror(fp)) fputc(*str++, fp);
}
```

Зависимые функции

fgetc(), fopen(), fprintf(), fread() и fwrite()



Функция fputs

```
#include <stdio.h>
int fputs(const char *str, FILE *stream);
```

Функция `fputs()` записывает в заданный поток *stream* содержимое строки, адресуемой указателем *str*. При этом завершающий нулевой символ (т.е. символ конца строки ('0')) не записывается.

В версии C99 к параметрам *str* и *stream* применен квалификатор `restrict`.

При успешном выполнении функция `fputs()` возвращает неотрицательное значение, а при неудачном — значение EOF.

Если поток открыт в текстовом режиме, могут произойти преобразования некоторых символов. Это значит, что однозначного отображения строки в файл может и не быть. Однако если поток открыт в двоичном режиме, никаких преобразований символов не будет и строка отобразится в файл “один к одному”.

Пример

Приведенный фрагмент программы записывает в поток, связанный с файлом `fp`, строку это тест.

```
fputs("это тест", fp);
```

Зависимые функции

fgets(), gets(), puts(), fprintf() и fscanf()



Функция fread

```
#include <stdio.h>
size_t fread(void *buf, size_t size, size_t count, FILE *stream);
```

Функция `fread()` читает из потока, адресуемого указателем *stream*, *count* объектов длиной *size* байт и размещает их в массиве *buf*. Затем указатель текущей позиции файла увеличивается на число, равное прочитанному количеству символов.

В версии C99 к параметрам *buf* и *stream* применен квалификатор `restrict`.

Функция `fread()` возвращает число реально прочитанных элементов. Если оказалось, что прочитано меньше элементов, чем требовалось при вызове, значит, либо произошла ошибка при выполнении операции, либо был достигнут конец файла. Определить, что именно произошло, можно с помощью функции `feof()` или `ferror()`.

Если поток открывается для операций в текстовом режиме, могут выполняться преобразования некоторых последовательностей символов, например, комбинация кодов возврата каретки (ASCII 13) и конца строки (ASCII 10) преобразуется в разделитель строк.

Пример

Следующая программа записывает в дисковый файл с названием TEST пять чисел с плавающей запятой, взяв их из массива `bal`. Затем она читает их из файла и записывает обратно в массив.

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

int main(void)
{
    FILE *fp;
    float bal[5] = { 1.1F, 2.2F, 3.3F, 4.4F, 5.5F };
    int i;

    /* запись значений */
    if((fp=fopen("test", "wb"))==NULL) {
        printf("Не удастся открыть файл.\n");
        exit(1);
    }

    if(fwrite(bal, sizeof(float), 5, fp) !=5)
        printf("Ошибка при записи файла.");
    fclose(fp);

    /* чтение значений */
    if((fp=fopen("test", "rb"))==NULL) {
        printf("Не удастся открыть файл.\n");
        exit(1);
    }

    if(fread(bal, sizeof(float), 5, fp) != 5) {
        if(feof(fp)) printf("Преждевременное достижение конца файла.");
        else printf("Ошибка при чтении файла.");
    }
    fclose(fp);

    for(i=0; i<5; i++)
        printf("%f", bal[i]);

    return 0;
}
```

Зависимые функции

`fwrite()`, `fopen()`, `fscanf()`, `fgetc()` и `getc()`



Функция `freopen`

```
#include <stdio.h>
FILE *freopen(const char *fname, const char *mode, FILE *stream);
```

Функция `freopen()` связывает существующий поток с другим файлом. Имя нового файла задается параметром *fname*, режим доступа — параметром *mode*, а переназначаемый поток определяется указателем *stream*. Возможные значения строки *mode* — те же, что и для функции `fopen()` (полное их описание можно найти в разделе, посвященном описанию `fopen()`).

В версии C99 к параметрам *fname*, *mode* и *stream* применен квалификатор `restrict`.

При вызове функция `freopen()` сначала пытается закрыть файл, который в данный момент связан с потоком *stream*. Но даже если этот файл закрыть не удастся, `freopen()` открывает другой файл.

При успешном выполнении функция `freopen()` возвращает указатель на поток, а в противном случае — нулевой указатель.

Чаще всего функция `freopen()` используется для перенаправления таких определенных системой файлов, как `stdin`, `stdout` и `stderr`, в какой-то другой.

Пример

Приведенная здесь программа использует функцию `freopen()`, чтобы перенаправить поток `stdout` в файл с названием `OUT`. Первое сообщение программы выводится на экран, а второе, перенаправленное, записывается функцией `freopen()`, в файл на диске.

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

int main(void)
{
    FILE *fp;

    printf("Это сообщение появится на дисплее.\n");

    if((fp=freopen("OUT", "w", stdout))==NULL) {
        printf("Не удастся открыть файл.\n");
        exit(1);
    }

    printf("Это сообщение будет записано в файл OUT.\n");

    fclose(fp);

    return 0;
}
```

Зависимые функции

`fopen()` и `fclose()`

Функция `fscanf`

```
#include <stdio.h>
int fscanf(FILE *stream, const char *format, ...);
```

Функция `fscanf()` работает подобно функции `scanf()`, но читает информацию не из стандартного потока ввода `stdin`, а из потока, заданного указателем *stream*. Подробности рассматриваются в разделе этой главы, посвященном функции `scanf`.

В версии C99 к параметрам *stream* и *format* применен квалификатор `restrict`.

Функция `fscanf()` возвращает количество аргументов, которым действительно присвоены значения. Это число не включает опущенные поля. Если возвращаемое функцией значение равно EOF, то это свидетельствует о том, что до выполнения первого присваивания произошел сбой.

Пример

Приведенный фрагмент программы читает из потока `fp` строку и значение переменной `f` с плавающей точкой (типа `float`).

```
char str[80];
float f;

fscanf(fp, "%s%f", str, &f);
```

Зависимые функции

`scanf()` и `fprintf()`



Функция `fseek`

```
#include <stdio.h>
int fseek(FILE *stream, long int offset, int origin);
```

Функция `fseek()` устанавливает указатель текущей позиции файла, связанного с потоком `stream`, в соответствии со значениями начала отсчета `origin` и смещения `offset`. Назначение этой функции — поддерживать операции ввода/вывода с произвольным доступом. Параметр `offset` равен количеству байтов, на которые будет смещен внутренний указатель файла относительно начала отсчета, заданного параметром `origin`. В качестве значения для параметра `origin` должен быть взят один из следующих макросов (определенных в заголовке `<stdio.h>`).

Имя	Назначение
SEEK_SET	Поиск с начала файла
SEEK_CUR	Поиск с текущей позиции
SEEK_END	Поиск с конца файла

Нулевое значение возврата свидетельствует об успешном выполнении функции `fseek()`, а ненулевое — о возникновении сбоя.

Вообще говоря, функцию `fseek()` следует использовать только при работе с двоичными файлами. При использовании же с текстовым файлом параметр `origin` должен иметь значение `SEEK_SET`, а параметр `offset` — значение, полученное в результате вызова функции `ftell()` для того же файла, или нуль (чтобы установить указатель текущей позиции файла на начало).

Функция `fseek()` очищает признак конца файла, связанный с заданным потоком. Более того, она аннулирует любой символ, ранее возвращенный в тот же поток через вызов функции `ungetc()` (см. раздел `ungetc`).

Пример

Приведенная функция производит поиск заданной структуры, имеющей тип `addr`. Обратите внимание на использование оператора `sizeof` для получения размера структуры.


```

struct addr {
    char name[40];
    char street[40];
    char city[40];
    char state[3];
    char zip[10];
} info;

void find(long int client_num)
{
    FILE *fp;

    if((fp=fopen("mail", "rb")) == NULL) {
        printf("Не удается открыть файл.\n");
        exit(1);
    }

    /* поиск подходящей структуры */
    fseek(fp, client_num*sizeof(struct addr), SEEK_SET);

    /* считывание данных в память */
    fread(&info, sizeof(struct addr), 1, fp);

    fclose(fp);
}

```

Зависимые функции

`ftell()`, `rewind()`, `fopen()`, `fgetpos()` и `fsetpos()`



Функция `fsetpos`

```

#include <stdio.h>
int fsetpos(FILE *stream, const fpos_t *position);

```

Функция `fsetpos()` перемещает указатель текущей позиции файла в место, заданное объектом, к которому отсылает указатель *position*. Это значение должно быть предварительно получено путем обращения к функции `fgetpos()`. После выполнения функции `fsetpos()` признак конца файла сбрасывается. Кроме того, аннулируется любой предыдущий результат обращения к функции `ungetc()`.

При неудачном выполнении функции `fsetpos()` возвращается ненулевое значение, а при успешном — нуль.

Пример

Данный фрагмент программы устанавливает указатель текущей позиции файла в новое положение, соответствующее значению переменной `file_loc`.

```

fsetpos(fp, &file_loc);

```

Зависимые функции

`fgetpos()`, `fseek()` и `ftell()`

Функция `ftell`

```
#include <stdio.h>
long int ftell(FILE *stream);
```

Функция `ftell()` возвращает текущее значение указателя позиции файла для заданного потока. В случае двоичных потоков это значение равно количеству байтов, которые отделяют указатель от начала файла. Для текстовых потоков возвращаемое значение может не иметь определенной интерпретации за исключением случая, когда оно является аргументом функции `fseek()`. Все дело в возможных преобразованиях символов, когда, например, комбинация кодов возврата каретки (ASCII 13) и конца строки (ASCII 10) заменяются разделителем строк, что влияет на размер файла.

При возникновении ошибки функция `ftell()` возвращает значение `-1`.

Пример

Данный фрагмент программы считывает текущее значение указателя позиции файла, связанного с потоком, который задается параметром `fp`.

```
long int i;

if((i=ftell(fp)) == -1L)
    printf("Возникла ошибка при обработке файла.\n")
```

Зависимые функции

`fseek()` и `fgetpos()`

Функция `fwrite`

```
#include <stdio.h>
size_t fwrite(const void *buf, size_t size, size_t count, FILE
*stream);
```

Функция `fwrite()` записывает в поток, адресуемый указателем `stream`, `count` объектов длиной `size` байтов каждый из массива символов, адресуемого указателем `buf`. Затем указатель текущей позиции файла перемещается вперед на записанное количество символов.

В версии C99 к параметрам `buf` и `stream` применен квалификатор `restrict`.

Функция `fwrite()` возвращает число реально записанных элементов, которое при успешном выполнении функции будет равно числу затребованных элементов. Если же элементов записано меньше, чем указано при вызове, произошла ошибка.

Пример

Данная программа записывает в файл `TEST` число с плавающей точкой (значение переменной `f`). Обратите внимание, что оператор `sizeof` используется и для определения количества байтов, занимаемых переменной с плавающей точкой, а также чтобы обеспечить переносимость.

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

int main(void)
{
```

```
FILE *fp;
float f=12.23;

if((fp=fopen("test", "wb"))==NULL) {
    printf("Не удается открыть файл.\n");
    exit(1);
}

fwrite(&f, sizeof(float), 1, fp);

fclose(fp);

return 0;
}
```

Зависимые функции

`fread()`, `fscanf()`, `getc()` и `fgetc()`

Функция `getc`

```
#include <stdio.h>
int getc(FILE *stream);
```

Функция `getc()` возвращает из входного потока *stream* символ, следующий за указателем текущей позиции, а затем увеличивает значение указателя текущей позиции. При чтении символа предполагается, что он имеет тип `unsigned char`, который потом преобразуется в целый.

При достижении конца файла функция `getc()` возвращает значение `EOF`. Но поскольку значение `EOF` само является целым значением, при работе с двоичными файлами проверять условие достижения конца файла необходимо с помощью функции `feof()`. При обнаружении ошибки функция `getc()` также возвращает значение `EOF`. Поэтому для выявления ошибок при работе с двоичными файлами необходимо использовать функцию `ferror()`.

Функции `getc()` и `fgetc()` идентичны, но в большинстве реализаций функция `getc()` определена как макрос.

Пример

Следующая программа читает содержимое текстового файла и выводит его на экран.

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

int main(int argc, char *argv[])
{
    FILE *fp;
    char ch;

    if((fp=fopen(argv[1], "r"))==NULL) {
        printf("Не удается открыть файл.\n");
        exit(1);
    }

    while((ch=getc(fp))!=EOF) {
```

```

    printf("%c", ch);
}

fclose(fp);

return 0;
}

```

Зависимые функции

fputc(), fgetc(), putc() и fopen()



Функция getchar

```

#include <stdio.h>
int getchar(void);

```

Функция `getchar()` возвращает из стандартного потока *stdin* следующий символ. При чтении символа предполагается, что символ имеет тип `unsigned char`, который потом преобразуется в целый.

При достижении конца файла, как и при обнаружении ошибки, функция `getchar()` возвращает значение `EOF`.

Функция `getchar()` чаще всего реализована как макрос.

Пример

Данная программа считывает в массив *s* символы из стандартного входного потока *stdin*, пока пользователь не нажмет клавишу `ENTER`. Затем введенная строка выводится на экран.

```

#include <stdio.h>

int main(void)
{
    char s[256], *p;

    p = s;

    while((*p++ = getchar()) != '\n');
    *p = '\0'; /* добавляем символ конца строки */
    printf(s);

    return 0;
}

```

Зависимые функции

fputc(), fgetc(), putc() и fopen()



Функция gets

```

#include <stdio.h>
char *gets(char *str);

```

Функция `gets()` читает символы из стандартного потока `stdin` и помещает их в массив символов, адресуемый указателем `str`. Символы читаются до тех пор, пока не встретится разделитель строк или значение EOF. Вместо разделителя строк в конец строки вставляется нулевой символ, свидетельствующий о ее завершении.

При успешном выполнении функция `gets()` возвращает указатель `str`, а при сбое — нулевой указатель. Если произошла ошибка, содержимое массива, адресуемого параметром `str`, не определено. Поскольку функция `gets()` возвращает нулевой указатель и при возникновении ошибки, и при достижении конца файла, то для выяснения, что же произошло на самом деле, необходимо использовать функцию `feof()` или `ferror()`.

Следует учесть, что нет способа ограничить число символов, которое прочитает функция `gets()`. Это означает, что массив, адресуемый указателем `str`, может переполниться. Следовательно, данная функция опасна по своей природе. Ее следует использовать только в пробных программах или утилитах “внутреннего” назначения, т.е. для себя. В коммерческих программах эту функцию использовать не рекомендуется.

Пример

В данной программе функция `gets()` используется для чтения названия файла.

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

int main(void)
{
    FILE *fp;
    char fname[128];

    printf("Введите имя файла: ");
    gets(fname);

    if((fp=fopen(fname, "r"))==NULL) {
        printf("Не удастся открыть файл.\n");
        exit(1);
    }

    fclose(fp);

    return 0;
}
```

Зависимые функции

`fputs()`, `fgetc()`, `fgets()` и `puts()`



Функция perror

```
#include <stdio.h>
void perror(const char *str);
```

Функция `perror()` преобразует значение глобальной переменной `errno` в строку и записывает эту строку в поток ошибок `stderr`. Если значение параметра `str` не равно нулю, то сначала записывается сама строка, за ней ставится двоеточие, а затем следует сообщение об ошибке, определяемое конкретной реализацией.

Пример

Этот фрагмент выдает сообщение о любой ошибке ввода/вывода, которая может произойти в потоке, связанном с файлом `fp`.

```
if(ferror(fp)) perror(" Ошибка при работе с файлом ");
```

Функция printf

```
#include <stdio.h>
int printf(const char *format, ...);
```

Функция `printf()` записывает в стандартный поток `stdout` значения аргументов из заданного списка аргументов в соответствии со строкой форматирования, адресуемой параметром *format*.

В версии C99 к параметру *format* применен квалификатор `restrict`.

Строка форматирования состоит из элементов двух типов. К элементам первого типа относятся символы, которые выводятся на экран. Элементы второго типа содержат спецификации формата, определяющие способ отображения аргументов. Спецификация формата начинается символом процента, за которым следует код формата. Количество аргументов должно в точности совпадать с количеством спецификаций формата, причем соответствие устанавливается в порядке их следования. Например, при вызове следующей функции `printf()` на экране будет отображено "Hi c 10 there!".

```
printf("Hi %c %d %s", 'c', 10, "there!");
```

Если заданных аргументов меньше, чем спецификаций формата, результат не определен. Если аргументов больше, чем спецификаций формата, оставшиеся аргументы отбрасываются. Спецификаторы формата перечислены в таблице 13.2.

Функция `printf()` возвращает число реально выведенных символов. Если функция возвратит отрицательное значение, то это будет свидетельствовать о наличии ошибки.

На спецификации формата могут воздействовать модификаторы, задающие ширину поля, точность и признак выравнивания по левому краю. Целое значение, расположенное между знаком `%` и командой форматирования, играет роль *спецификации минимальной ширины поля*. Наличие этого спецификатора приводит к тому, что результат будет заполнен пробелами или нулями, чтобы выводимое значение занимало поле, ширина которого не меньше заданной минимальной ширины. Если длина выводимого значения (строки или числа) больше этого минимума, оно будет выведено полностью несмотря на превышение минимума. По умолчанию в качестве заполнителя используется пробел. Для заполнения нулями перед спецификацией ширины поля нужно поместить `0`. Например, спецификация формата `%05d` дополнит нулями выводимое число, в котором менее пяти цифр, чтобы общая длина равнялась 5 символам.

Действие *модификатора точности* зависит от кода формата, к которому он применяется. Чтобы добавить модификатор точности, поставьте за спецификацией ширины поля десятичную точку, а после нее — требуемое значение точности. Для форматов `a`, `A`, `e`, `E`, `f` и `F` модификатор точности определяет число выводимых десятичных знаков. Например, спецификация формата `%10.4f` обеспечит вывод числа с четырьмя знаками после запятой в поле шириной не меньше десяти символов. Если модификатор точности применяется к коду формата `g` или `G`, то он определяет максимальное число выводимых значащих цифр. Применительно к целым, модификатор точности задает минимальное количество выводимых цифр. При необходимости перед числом будут добавлены нули.

Если модификатор точности применяется к строкам, число, следующее за точкой, задает максимальную длину поля. Например, спецификация формата `%5.7s` выведет

строку длиной не менее пяти, но не более семи символов. Если выводимая строка окажется длиннее максимальной длины поля, конечные символы будут отсечены.

По умолчанию все выводимые значения выравниваются по правому краю: если ширина поля больше выводимого значения, оно будет выровнено по правому краю поля. Чтобы установить выравнивание по левому краю, нужно поставить знак “минус” сразу после знака %. Например, спецификация формата `%-10.2f` обеспечит выравнивание вещественного числа с двумя десятичными знаками в 10-символьном поле по левому краю.

Существуют два модификатора формата, позволяющие функции `printf()` отображать короткие и длинные целые. Эти модификаторы могут применяться к спецификаторам типа `d`, `i`, `o`, `u`, `x` и `X`. Модификатор `l` уведомляет функцию `printf()` о длинном типе значения. Например, спецификация `%ld` означает, что выводится длинное целое число. Модификатор `h` сообщает функции `printf()`, что нужно вывести число короткого целого типа. Следовательно, строка `%hu` означает, что выводимое данное имеет тип `short unsigned int`.

Таблица 13.2. Спецификаторы формата функции `printf()`

Код	Формат
<code>%a</code>	Выводит шестнадцатеричное число в форме <code>0xh.hhhhp+d</code> (только C99)
<code>%A</code>	Выводит шестнадцатеричное число в форме <code>0Xh.hhhhP+d</code> (только C99)
<code>%c</code>	Символ
<code>%d</code>	Десятичное целое число со знаком
<code>%i</code>	Десятичное целое число со знаком
<code>%e</code>	Экспоненциальное представление числа (в виде мантиссы и порядка) (е на нижнем регистре)
<code>%E</code>	Экспоненциальное представление числа (в виде мантиссы и порядка) (Е на верхнем регистре)
<code>%f</code>	Десятичное число с плавающей точкой
<code>%F</code>	Десятичное число с плавающей точкой (только C99; если применяется к бесконечности или к нечисловому значению, то выдает надписи <code>INF</code> , <code>INFINITY</code> или <code>NAN</code> на верхнем регистре. Спецификатор <code>%f</code> выводит их эквиваленты на нижнем регистре.)
<code>%g</code>	Использует более короткий из форматов <code>%e</code> или <code>%f</code>
<code>%G</code>	Использует более короткий из форматов <code>%E</code> или <code>%F</code>
<code>%o</code>	Восьмеричное число без знака
<code>%s</code>	Символьная строка
<code>%u</code>	Десятичное целое число без знака
<code>%x</code>	Шестнадцатеричное без знака (строчные буквы)
<code>%X</code>	Шестнадцатеричное без знака (прописные буквы)
<code>%p</code>	Выводит указатель
<code>%n</code>	Соответствующий аргумент должен быть указателем на целое число. (Этот спецификатор указывает, что в целочисленной переменной, на которую указывает ассоциированный с данным спецификатором указатель, будет храниться число символов, выведенных к моменту обработки спецификации <code>%n</code> .)
<code>%%</code>	Выводит знак процента

При использовании современного компилятора, поддерживающего добавленные в 1995 году средства работы с двухбайтовыми символами, можно к спецификации `s` применить модификатор `l`, чтобы указать на использование двухбайтовых символов. Модификатор `l` можно также использовать с командой формата `s` для вывода строки двухбайтовых символов.

Кроме того, модификатор `l` можно поставить перед командами форматирования вещественных чисел `a`, `A`, `e`, `E`, `f`, `F`, `g` и `G`. В этом случае он уведомит о выводе значения типа `long double`.

Команда `n` сохраняет в целой переменной, указатель на которую задан в списке аргументов, число символов, которые были записаны в поток вывода к моменту обнаружения спецификатора `n`. Например, следующий фрагмент программы после строки “Это тест” выведет число 8.

```
int i;  
  
printf("Это тест\n", &i);  
printf("%d", i);
```

Чтобы обозначить, что соответствующий аргумент указывает на длинное целое, к спецификации `n` можно применить модификатор `l`. Для указания на короткое целое примените к спецификации `n` модификатор `h`.

Символ `#` при использовании с некоторыми кодами формата функции `printf()` приобретает специальное значение. Поставленный перед кодами `a`, `A`, `g`, `G`, `f`, `e` и `E`, он гарантирует наличие десятичной точки даже в случае отсутствия десятичных цифр. Если поставить символ `#` перед кодами формата `x` и `X`, то шестнадцатеричное число будет выведено с префиксом `0x`. Если же его поставить перед кодами формата `o` и `O`, то восьмеричное число будет выведено с префиксом `0`. Символ `#` нельзя применять ни к каким другим спецификациям формата.

Спецификации минимальной ширины поля и точности могут задаваться не константами, а аргументами функции `printf()`. Для этого в строке форматирования используется символ “звездочка” (*). При сканировании строки форматирования функции `printf()` каждый символ * будет сопоставляться с соответствующими аргументами в порядке их следования.

Модификаторы формата функции `printf()`, добавленные стандартом C99

В версии C99 для использования в функции `printf()` добавлены модификаторы формата `hh`, `ll`, `j`, `z` и `t`. Модификатор `hh` можно применять к спецификаторам преобразования `d`, `i`, `o`, `u`, `x`, `X` и `n`. Он означает, что соответствующий аргумент является значением типа `signed char` или `unsigned char`, а в случае спецификации `n` — указателем на переменную типа `signed char`. Модификатор `ll` также можно применять к спецификаторам преобразования `d`, `i`, `o`, `u`, `x`, `X` и `n`. Он означает, что соответствующий аргумент является значением типа `signed long long int` или `unsigned long long int`, а в случае спецификатора `n` — указателем на переменную типа `long long int`. Версия C99 также позволяет применять модификатор `l` к спецификаторам преобразования чисел с плавающей точкой `a`, `A`, `e`, `E`, `f`, `F`, `g`, и `G`, но это не дает никакого эффекта.

Применение модификатора формата `j` к спецификаторам преобразования `d`, `i`, `o`, `u`, `x`, `X` и `n` устанавливает для соответствующего аргумента тип `intmax_t` или `uintmax_t`. Эти типы объявлены в заголовке `<stdint.h>` и служат для хранения целых самой большой разрядности.

Применение к спецификаторам преобразования `d`, `i`, `o`, `u`, `x`, `X` и `n` модификатора формата `z` устанавливает для соответствующего аргумента тип `size_t`. Этот тип объявлен в заголовке `<stddef.h>` и служит для хранения результата выполнения оператора `sizeof`.

Применение к спецификаторам преобразования `d`, `i`, `o`, `u`, `x`, `X` и `n` модификатора формата `t` устанавливает для соответствующего аргумента тип `ptrdiff_t`. Этот тип объявлен в заголовке `<stddef.h>` и служит для хранения значения разности между двумя указателями.

Пример

Данная программа выводит то, что указано в комментариях.

```
#include <stdio.h>

int main(void)
{
    /* Этот фрагмент печатает строку "это тест",
       которая выравнивается по левому краю поля шириной в 20 символов.
    */
    printf("%-20s", "это тест");

    /* Этот фрагмент печатает в поле шириной в 10 символов число
       с плавающей точкой с тремя десятичными разрядами после запятой.
       В результате получится "    12.235".
    */
    printf("%10.3f", 12.234657);

    return 0;
}
```

Зависимые функции

scanf() и fprintf()



Функция puts

```
#include <stdio.h>
int puts(int ch, FILE *stream);
```

Функция `puts()` записывает в поток вывода, адресуемый параметром *stream*, символ, содержащийся в младшем байте параметра *ch*. Поскольку в момент вызова символьные аргументы преобразуются в целые, их вполне можно использовать в качестве аргументов функции `puts()`. Функция `puts()` часто реализуется как макрос.

При успешном выполнении функция `puts()` возвращает записанный символ, а в случае ошибки — значение EOF. Если поток вывода был открыт в двоичном режиме, EOF тоже может быть воспринято как *ch*. Поэтому в данном случае для выявления ошибки необходимо использовать функцию `ferror()`.

Пример

Следующий цикл записывает символы в строку *str* потока, заданного идентификатором *fp*. Символ конца строки не записывается.

```
for(; *str; str++) puts(*str, fp);
```

Зависимые функции

fgetc(), fputc(), getchar() и putchar()



Функция putchar

```
#include <stdio.h>
int putchar(int ch);
```

Функция `putchar()` записывает символ, содержащийся в младшем байте параметра *ch*, в стандартный поток вывода `stdout`. По выполняемому действию она эквивалентна `putc(ch, stdout)`. Поскольку в момент вызова символьные аргументы преобразуются к целому типу, их вполне можно использовать в качестве аргументов функции `putchar()`.

При успешном выполнении функция `putchar()` возвращает записанный символ, а в случае ошибки — значение `EOF`.

Пример

Следующий цикл записывает символы в строку *str* стандартного потока вывода `stdout`. Символ конца строки не записывается.

```
for(; *str; str++) putchar(*str);
```

Зависимые функции

`putc()`



Функция puts

```
#include <stdio.h>
int puts(const char *str);
```

Функция `puts()` записывает строку, адресуемую параметром *str*, в стандартное выходное устройство. Символ конца строки преобразуется в разделитель строк.

При успешном выполнении функция `puts()` возвращает неотрицательное значение, а в случае сбоя — значение `EOF`.

Пример

Следующая программа записывает в стандартный поток вывода `stdout` строку *это пример*.

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

int main(void)
{
    char str[80];

    strcpy(str, "это пример");

    puts(str);

    return 0;
}
```

Зависимые функции

`putc()`, `gets()` и `printf()`

Функция remove

```
#include <stdio.h>
int remove(const char *fname);
```

Функция `remove()` удаляет файл, заданный параметром *fname*. При успешном удалении файла функция возвращает нуль, а в случае ошибки — ненулевое значение.

Пример

Данная программа удаляет файл, имя которого задается в командной строке.

```
#include <stdio.h>

int main(int argc, char *argv[])
{
    if(remove(argv[1])) printf("Ошибка при удалении");

    return 0;
}
```

Зависимые функции

`rename()`

Функция rename

```
#include <stdio.h>
int rename(const char *oldfname, const char *newfname);
```

Функция `rename()` переименовывает файл; она заменяет имя файла, заданное параметром *oldfname*, именем, заданным параметром *newfname*. Имя, заданное параметром *newfname*, не должно совпадать ни с одним из существующих в каталоге имен файлов.

При успешном выполнении функция `rename()` возвращает нуль, а в случае ошибки — ненулевое значение.

Пример

Данная программа заменяет имя файла, заданное первым (нумерация аргументов начинается с нуля! — *Прим. ред.*) аргументом командной строки, именем, которое задается вторым аргументом командной строки. Учитывая, что программа называется `CHANGE`, командная строка

`CHANGE THIS THAT`

приведет к переименованию файла `THIS` в файл `THAT`.

```
#include <stdio.h>

int main(int argc, char *argv[])
{
    if(rename(argv[1], argv[2]) !=0)
        printf("Ошибка при переименовании");

    return 0;
}
```

Зависимые функции

`remove()`

Функция `rewind`

```
#include <stdio.h>
void rewind(FILE *stream);
```

Функция `rewind()` перемещает указатель текущей позиции файла в начало заданного потока. Она также очищает связанные с потоком *stream* признаки конца файла и ошибок.

Пример

Данная функция дважды читает поток, адресованный указателем `fp`, и каждый раз выводит файл на экран.

```
void re_read(FILE *fp)
{
    /* первое чтение */
    while(!feof(fp)) putchar(getc(fp));

    rewind(fp);

    /* второе чтение */
    while(!feof(fp)) putchar(getc(fp));
}
```

Зависимые функции

`fseek()`

Функция `scanf`

```
#include <stdio.h>
int scanf(const char *format, ...);
```

Функция `scanf()` представляет собой процедуру ввода общего назначения, которая читает поток `stdin` и сохраняет информацию в переменных, перечисленных в списке аргументов. Она может читать все встроенные типы данных и автоматически преобразовывать их в соответствующий внутренний формат.

В версии C99 к параметру *format* применен квалификатор `restrict`.

Управляющая строка, задаваемая параметром *format*, состоит из символов трех категорий:

- спецификаторов формата;
- пробельных символов;
- символов, отличных от пробельных.

Спецификации формата начинаются знаком `%` и сообщают функции `scanf()` тип данного, которое будет прочитано. Спецификации формата приведены в таблице 13.3. Например, по спецификации `%s` будет прочитана строка, а по спецификации `%d` —

целое значение. Строка форматирования читается слева направо, и спецификации формата сопоставляются аргументам в порядке их перечисления в списке аргументов.

Таблица 13.3. Спецификации формата функции `scanf()`

Код	Назначение
<code>%a</code>	Читает значение с плавающей точкой (только C99)
<code>%A</code>	Аналогично коду <code>%a</code> (только C99)
<code>%c</code>	Читает один символ
<code>%d</code>	Читает десятичное целое
<code>%i</code>	Читает целое в любом формате (десятичное, восьмеричное или шестнадцатеричное)
<code>%e</code>	Читает число с плавающей точкой
<code>%E</code>	Аналогично коду <code>%e</code>
<code>%f</code>	Читает число с плавающей точкой
<code>%F</code>	Аналогично коду <code>%f</code> (только C99)
<code>%g</code>	Читает число с плавающей точкой
<code>%G</code>	Аналогично коду <code>%g</code>
<code>%o</code>	Читает восьмеричное число
<code>%s</code>	Читает строку
<code>%x</code>	Читает шестнадцатеричное число
<code>%X</code>	Аналогично коду <code>%x</code>
<code>%p</code>	Читает указатель
<code>%n</code>	Принимает целое значение, равное количеству прочитанных до сих пор символов
<code>%u</code>	Читает десятичное целое без знака
<code>%[]</code>	Просматривает набор символов
<code>%%</code>	Читает знак процента

По умолчанию спецификации `a`, `f`, `e` и `g` заставляют функцию `scanf()` присваивать данные переменным типа `float`. Если перед одной из этих спецификаций поставить модификатор `l`, функция `scanf()` присвоит прочитанные данные переменной типа `double`. Использование же модификатора `L` означает, что полученное значение присвоится переменной типа `long double`.

Современные компиляторы, поддерживающие добавленные в 1995 году средства работы с двухбайтовыми символами, позволяют к спецификации `s` применить модификатор `l`; тогда будет считаться, что соответствующий указатель указывает на двухбайтовый символ (т.е. на данное типа `wchar_t`). Модификатор `l` также можно использовать с кодом формата `s`; тогда будет считаться, что соответствующий указатель указывает на строку двухбайтовых символов. Кроме того, модификатор `l` можно использовать для того, чтобы указать, что набор сканируемых символов состоит из двухбайтовых символов.

Если в строке форматирования встретится разделитель, то функция `scanf()` пропустит один или несколько разделителей во входном потоке. Под разделителем, или пробельным символом, подразумевается пробел, символ табуляции или разделитель строк (символ новой строки). По сути, наличие одного разделителя в управляющей строке приведет к тому, что функция `scanf()` будет читать, не сохраняя, любое количество (возможно, даже нулевое) разделителей до первого символа, отличного от разделителя.

Если в строке форматирования встретился символ, отличный от разделителя, то функция `scanf()` прочитает и отбросит его. Например, если в строке форматирования встретится `%d,%d`, то функция `scanf()` сначала прочитает целое значение, затем прочитает и отбросит запятую и, наконец, прочитает еще одно целое. Если заданный символ не найден, функция `scanf()` завершает работу.

Все переменные, получающие значения с помощью функции `scanf()`, должны передаваться посредством своих адресов. Это значит, что все аргументы должны быть указателями на переменные.

Элементы входного потока должны быть разделены пробелами, символами табуляции или разделителями строк. Такие символы, как запятая, точка с запятой и т.п., не распознаются в качестве разделителей. Это означает, что оператор

```
scanf("%d%d", &r, &c);
```

примет значения, введенные как 10 20, но откажется от последовательности символов 10,20.

Символ `*`, стоящий после знака `%` и перед кодом формата, прочитает данные заданного типа, но запретит их присваивание. Следовательно, оператор

```
scanf("%d*%c%d", &x, &y);
```

при вводе данных в виде 10/20 поместит значение 10 в переменную `x`, отбросит знак деления и присвоит значение 20 переменной `y`.

Команды форматирования могут содержать модификатор максимальной длины поля. Он представляет собой целое число, располагаемое между знаком `%` и кодом формата, которое ограничивает количество читаемых для всех полей символов. Например, если в переменную `address` нужно прочитать не более 20 символов, используется следующий оператор.

```
scanf("%20s", address);
```

Если входной поток содержит более 20 символов, то при последующем обращении к операции ввода чтение начнется с того места, в котором “остановился” предыдущий вызов функции `scanf()`. Если разделитель встретится раньше, чем достигнута максимальная длина поля, ввод данных завершится. В этом случае функция `scanf()` переходит к чтению следующего поля.

Хотя пробелы, символы табуляции и разделители строк используются в качестве разделителей полей, при чтении одиночного символа они читаются подобно любому другому символу. Например, если входной поток состоит из символов `x y`, то оператор

```
scanf("%c%c%c", &a, &b, &c);
```

поместит символ `x` в переменную `a`, пробел — в переменную `b`, а символ `y` — в переменную `c`.

Помните, что любые символы управляющей строки (включая пробелы, символы табуляции и новой строки), не являющиеся спецификациями формата, используются для установки соответствия и отбрасывания символов из входного потока. Любой соответствующий им символ отбрасывается. Например, если поток ввода выглядит, как 10t20, оператор

```
scanf("%dt%d", &x, &y);
```

присвоит переменной `x` значение 10, а переменной `y` — значение 20. Символ `t` отбрасывается, так как он присутствует в управляющей строке.

Функция `scanf()` поддерживает спецификатор формата общего назначения, называемый *набором сканируемых символов* (*scanset*). В этом случае определяется набор символов, которые могут быть прочитаны функцией `scanf()` и присвоены соответствующему массиву символов. Для определения такого набора символы, подлежащие сканированию, необходимо заключить в квадратные скобки. Открывающая квадратная скобка должна следовать сразу за знаком процента. Например, следующий набор сканируемых символов указывает на то, что необходимо читать только символы `A`, `B` и `C`.

```
scanf("%[ABC]
```

При использовании набора сканируемых символов функция `scanf()` продолжает читать символы и помещать их в соответствующий массив символов до тех пор, пока не встретится символ, отсутствующий в заданном наборе. Соответствующая набору переменная должна быть указателем на массив символов. При возврате из функции `scanf()` этот массив будет содержать строку из прочитанных символов, завершающуюся символом конца строки.

Если первый символ в наборе является знаком `^`, то получаем обратный эффект: входное поле читается до тех пор, пока не встретится символ из заданного набора сканируемых символов, т.е. знак `^` заставляет функцию `scanf()` читать только те символы, которые отсутствуют в наборе сканируемых символов.

Во многих реализациях допускается задавать диапазон с помощью дефиса. Например, функция `scanf()`, встречая набор сканируемых символов в виде `%[A-Z]`, будет читать символы, попадающие в диапазон от `A` до `Z`.

Важно помнить, что в наборе сканируемых символов различаются прописные и строчные буквы. Следовательно, чтобы сканировать как прописные, так и строчные буквы, в наборе сканируемых символов придется задать их отдельно.

Функция `scanf()` возвращает число, равное количеству полей, для которых успешно присвоены значения. К этим полям не относятся поля, которые были прочитаны, но присвоение не состоялось в связи с использованием модификатора `*`, подавляющего присваивание. При обнаружении ошибки до присвоения значения первого поля функция `scanf()` возвращает значение `EOF`.

Модификаторы формата, добавленные к функции `scanf()` Стандартом C99

В версии C99 для использования в функции `scanf()` добавлены модификаторы формата `hh`, `ll`, `j`, `z` и `t`. Модификатор `hh` можно применять к спецификациям `d`, `i`, `o`, `u`, `x` и `n`. Он означает, что соответствующий аргумент является указателем на значение типа `signed char` или `unsigned char`. Модификатор `ll` также можно применять к спецификациям `d`, `i`, `o`, `u`, `x` и `n`. Он означает, что соответствующий аргумент является указателем на значение типа `signed long long int` или `unsigned long long int`.

Модификатор формата `j`, который применяется к спецификациям `d`, `i`, `o`, `u`, `x` и `n`, означает, что соответствующий аргумент является указателем на значение типа `intmax_t` или `uintmax_t`. Эти типы объявлены в заголовке `<stdint.h>` и служат для хранения целых максимально возможной разрядности.

Модификатор формата `z`, который применяется к спецификациям `d`, `i`, `o`, `u`, `x` и `n`, означает, что соответствующий аргумент является указателем на объект типа `size_t`. Этот тип объявлен в заголовке `<stddef.h>` и служит для хранения результата операции `sizeof`.

Модификатор формата `t`, который применяется к спецификациям `d`, `i`, `o`, `u`, `x` и `n`, означает, что соответствующий аргумент является указателем на объект типа `ptrdiff_t`. Этот тип объявлен в заголовке `<stddef.h>` и служит для хранения значения разности между двумя указателями.

Пример

Действие данных операторов `scanf()` объясняется в комментариях.

```
#include <stdio.h>

int main(void)
{
```

```

char str[80], str2[80];
int i;

/* читается строка и целое значение */
scanf("%s%d", str, &i);

/* в переменную str считывается не более 79 символов */
scanf("%79s", str);

/* целое, расположенное между двумя строками, пропускается */
scanf("%s%d%s", str, str2);

return 0;
}

```

Зависимые функции

printf() и fscanf()



Функция setbuf

```

#include <stdio.h>
void setbuf(FILE *stream, char *buf);

```

Функция `setbuf()` задает буфер, которым будет пользоваться поток *stream*, либо отключает буферизацию, если параметр *buf* установлен равным нулю. Если необходимо задать буфер, определенный программистом, его длину следует установить равной `BUFSIZ` символом. Идентификатор `BUFSIZ` определяется в заголовке `<stdio.h>`.

В версии C99 к параметрам *stream* и *buf* применен квалификатор `restrict`.

Пример

Следующий фрагмент связывает буфер, определенный программистом, с потоком, адресуемым указателем `fp`.

```

char buffer[BUFSIZ];
.
.
.
setbuf(fp, buffer);

```

Зависимые функции

fopen(), fclose() и setvbuf()



Функция setvbuf

```

#include <stdio.h>
int setvbuf(FILE *stream, char *buf, int mode, size_t size);

```

Функция `setvbuf()` позволяет программисту задать буфер, его размер и режим работы с указанным потоком. Массив символов, адресуемый параметром *buf*, используется в качестве буфера потока для операций ввода/вывода. Размер буфера устанавливается с помощью параметра *size*, а режим *mode* определяет, как будет

выполняться буферизация. Если параметр *buf* равен нулю, функция `setvbuf()` выделяет собственный буфер.

В версии C99 к параметрам *stream* и *buf* применен квалификатор `restrict`.

Возможными значениями параметра *mode* являются `_IOFBF`, `_IONBF` и `_IOLBF`, которые определены в заголовочном файле `<stdio.h>`. Если параметр *mode* равен `_IOFBF`, для буферизации используется полный объем буфера. Если *mode* равен `_IOLBF`, поток будет буферизирован построчно, т.е. содержимое буфера будет дозаписываться в поток при каждой записи в поток вывода символа новой строки. Содержимое буфера также дозаписывается в поток при заполнении буфера. При чтении из входного потока появление разделителя строк приведет к прекращению подкачки в буфер. Если установлен режим `_IONBF`, поток не буферизируется.

Функция `setvbuf()` возвращает нуль при успешном выполнении, а в противном случае ненулевое значение.

Пример

Данный фрагмент программы устанавливает построчный режим вывода потока *fp* в буфер размером 128 символов.

```
#include <stdio.h>
char buffer[128];
.
.
.
setvbuf(fp, buffer, _IOLBF, 128);
```

Зависимые функции

`setbuf()`

Функция `snprintf`

```
#include <stdio.h>
int snprintf(char * restrict buf, size_t num, const char * restrict
format,...);
```

Функция `snprintf()` добавлена в версии C99.

Она идентична функции `sprintf()` за исключением того, что в массиве, адресуемом указателем *buf*, будет сохранено максимум *num-1* символов. По окончании работы функции этот массив будет завершаться символом конца строки (нуль-символом). Таким образом, функция `snprintf()` позволяет предотвратить переполнение буфера *buf*.

Зависимые функции

`printf()`, `sprintf()` и `fsprintf()`

Функция `sprintf`

```
#include <stdio.h>
int sprintf(char *buf, const char *format,...);
```

Функция `sprintf()` идентична функции `printf()` за исключением того, что поток вывода записывается в массив, адресуемый указателем *buf*, а не в стандартный по-

ток stdout. По окончании работы функции этот массив будет завершаться символом конца строки (нуль-символом). Подробности рассматриваются в разделе, посвященном описанию функции printf.

В версии C99 к параметрам *buf* и *format* применен квалификатор restrict.

Возвращаемое значение равно числу символов, действительно помещенных в массив.

Важно понимать, что функция sprintf() не обеспечивает никакой проверки переполнения массива, адресуемого указателем *buf*. Это значит, что массив будет переполнен, если объем выводимых символов превысит длину массива. В качестве альтернативного решения рассмотрите применение функции snprintf().

Пример

После выполнения этого фрагмента программы элементам массива str значения будут присвоены таким образом, что получится строка один 2 3.

```
char str[80];  
sprintf(str, "%s %d %c", "один", 2, '3');
```

Зависимые функции

printf() и fprintf()



Функция sscanf

```
#include <stdio.h>  
int sscanf(const char *buf, const char *format, ...);
```

Функция sscanf() идентична функции scanf(), но данные читаются из массива, адресуемого параметром *buf*, а не из стандартного потока ввода stdin. Подробности приводятся в разделе scanf.

В версии C99 к параметрам *buf* и *format* применен квалификатор restrict.

Значение, возвращаемое функцией, равно количеству переменных, которым реально были присвоены значения. К ним не относятся поля, опущенные из-за использования модификатора команды форматирования *. Нулевое значение свидетельствует о том, что ни одно поле не было присвоено, а значение EOF сигнализирует об ошибке, обнаруженной до первого присваивания.

Пример

Данная программа выводит на экран сообщение привет 1.

```
#include <stdio.h>  
  
int main(void)  
{  
    char str[80];  
    int i;  
  
    sscanf("привет 1 2 3 4 5", "%s%d", str, &i);  
    printf("%s %d", str, i);  
  
    return 0;  
}
```

Зависимые функции

`scanf()` и `fscanf()`



Функция `tmpfile`

```
#include <stdio.h>
FILE *tmpfile(void);
```

Функция `tmpfile()` открывает временный двоичный файл для операций чтения-записи и возвращает указатель на связанный с ним поток. Она автоматически использует уникальное имя файла, чтобы избежать конфликтов с существующими файлами.

Функция `tmpfile()` при неудачном выполнении возвращает нулевой указатель, а при успешном — указатель на поток.

Временный файл, созданный функцией `tmpfile()`, автоматически удаляется при закрытии файла или по завершении программы.

Количество временных файлов, которые можно открыть, равно значению `TMP_MAX` (которое не превышает предел, определяемый значением `FOPEN_MAX`).

Пример

Следующий фрагмент создает временный файл.

```
FILE *temp;

if((temp=tmpfile())==NULL) {
    printf("Не удастся открыть временный файл.\n");
    exit(1);
}
```

Зависимые функции

`tmpnam()`



Функция `tmpnam`

```
#include <stdio.h>
char *tmpnam(char *name);
```

Функция `tmpnam()` генерирует уникальное имя файла и сохраняет его в массиве, адресуемом указателем `name`. Длина этого массива должна составлять не меньше `L_tmpnam` символов. (Константа `L_tmpnam` определена в заголовочном файле `<stdio.h>`.) Основное назначение функции `tmpnam()` — сгенерировать имя временного файла, которое не совпадало бы ни с одним из имен файлов в текущем каталоге диска.

Эту функцию можно вызвать не более `TMP_MAX` раз. Константа `TMP_MAX` определена в заголовочном файле `<stdio.h>`, и ее значение больше либо равно 25. При каждом вызове функция `tmpnam()` будет генерировать новое имя временного файла.

При успешном выполнении функция возвращает указатель на массив `name`, в противном случае — нулевой указатель. Если значение параметра `name` равно нулю, имя временного файла сохраняется в статическом массиве, принадлежащем функции `tmpnam()`, которая в этом случае возвращает указатель на этот массив. При последующем вызове функции `tmpnam()` этот массив будет перезаписан.

Пример

В данной программе генерируются и выводятся на экран три уникальных имени временных файлов.

```
#include <stdio.h>

int main(void)
{
    char name[40];
    int i;

    for(i=0; i<3; i++) {
        tmpnam(name);
        printf("%s ", name);
    }

    return 0;
}
```

Зависимые функции

tmpfile()



Функция ungetc

```
#include <stdio.h>
int ungetc(int ch, FILE *stream);
```

Функция `ungetc()` возвращает в поток ввода `stream` символ, заданный младшим байтом параметра `ch`. Затем этот символ будет получен при последующей операции чтения потока `stream`. Обращение к таким функциям, как `fflush()`, `fseek()` и `rewind()`, аннулирует действие `ungetc()` и сбрасывает этот символ.

Гарантируется, что в поток можно вернуть один символ, однако некоторые реализации допускают возврат большего числа символов.

Попытка вернуть в поток ввода значение EOF игнорируется.

Обращение к функции `ungetc()` очищает признак конца файла, связанный с заданным потоком. Значение указателя текущей позиции файла для текстового потока не определено до тех пор, пока не будут прочитаны все возвращенные обратно в поток символы, в этом случае оно остается таким же, каким было до первого вызова функции `ungetc()`. При работе с потоками в двоичном режиме каждый вызов функции `ungetc()` уменьшает указатель текущей позиции файла.

При успешном завершении функция возвращает значение `ch`, в противном случае — значение EOF.

Пример

Данная функция читает слова из входного потока, адресуемого указателем `fp`. Разделитель возвращается в поток для последующего использования. Например, если входные данные имеют вид `count/10`, то при первом обращении к функции `read_word()` она возвратит `count`, а символ `“/”` направит обратно во входной поток.

```
void read_word(FILE *fp, char *token)
{
    while(isalpha(*token=getc(fp))) token++;
    ungetc(*token, fp);
}
```

Зависимые функции

getc()

Функции vprintf, fprintf, vsprintf и vsnprintf

```
#include <stdarg.h>
#include <stdio.h>
int vprintf(char *format, va_list arg_ptr);
int fprintf(FILE *stream, const char *format, va_list arg_ptr);
int vsprintf(char *buf, const char *format, va_list arg_ptr);
int vsnprintf(char * restrict buf, size_t num,
              const char * restrict format, va_list arg_ptr);
```

Действия функций vprintf(), fprintf(), vsprintf() и vsnprintf() эквивалентны действиям функций printf(), fprintf(), sprintf() и snprintf() соответственно, но список аргументов заменяется указателем на список аргументов. Этот указатель должен иметь тип va_list, который определен в заголовке <stdarg.h>.

В версии C99 к параметрам *buf* и *format* применен квалификатор restrict. Функция vsnprintf() добавлена в версии C99.

Пример

Данный фрагмент программы иллюстрирует, как нужно вызывать функцию vprintf(). Вызов функции va_start() приводит к созданию указателя на список аргументов переменной длины, причем этот указатель указывает на начало списка аргументов. Этот указатель должен быть использован при вызове функции vprintf(). Вызов функции va_end() очищает указатель на список аргументов переменной длины.

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

void print_message(char *format, ...);

int main(void)
{
    print_message("Не удастся открыть файл %s.", "test");

    return 0;
}

void print_message(char *format, ...)
{
    va_list ptr; /* извлечение аргумента ptr */

    /* инициализация ptr, он становится указателем на первый аргумент,
       следующий за строкой форматирования */
    va_start(ptr, format);

    /* вывод сообщения */
    vprintf(format, ptr);

    va_end(ptr);
}
```

Зависимые функции

`vscanf()`, `vfscanf()`, `vsscanf()`, `va_arg()`, `va_start()` и `va_end()`



Функции `vscanf`, `vfscanf` и `vsscanf`

```
#include <stdarg.h>
#include <stdio.h>
int vscanf(char * restrict format, va_list arg_ptr);
int vfscanf(FILE * restrict stream, const char * restrict format,
            va_list arg_ptr);
int vsscanf(char * restrict buf, const char * restrict format,
            va_list arg_ptr);
```

Эти функции добавлены в версии C99.

Действия функций `vscanf()`, `vfscanf()` и `vsscanf()` эквивалентны действиям функций `scanf()`, `fscanf()` и `sscanf()` соответственно, но список аргументов заменен указателем на этот список. Данный указатель должен иметь тип `va_list`, который определен в заголовке `<stdarg.h>`.

Зависимые функции

`vprintf()`, `vfprintf()`, `vsprintf()`, `vsnprintf()`, `va_arg()`, `va_start()` и `va_end()`

Полный
справочник по



Глава 14

**Строковые и символьные
функции**

Стандартная библиотека языка С обладает богатым и разнообразным набором функций для обработки строк и символов. Строковые функции работают с массивами символов (строками), заканчивающимися символом конца строки. В языке С для работы со строковыми функциями используется заголовок `<string.h>`, для символьных функций — заголовочный файл `<ctype.h>`.

Поскольку в С не предусмотрен автоматический контроль нарушения границ массивов, вся ответственность за их переполнение ложится на программиста. Не следует этим пренебрегать, так как при переполнении массива может произойти аварийное завершение программы.

В С *печатаемыми символами* являются те, которые можно отобразить на терминале. В ASCII-средах они расположены между пробелом (0x20) и тильдой (0xFE). *Управляющие символы* имеют значения, лежащие в диапазоне между нулем и 0x1F; в ASCII-средах к ним также относится символ DEL (0x7F).

Исторически сложилось так, что аргументами символьных функций являются целые значения, из которых используется только младший байт. Символьные функции автоматически преобразуют свои аргументы в тип `unsigned char`. Безусловно, эти функции можно вызывать с символьными аргументами, поскольку в момент вызова функции символы автоматически преобразуются к целому типу.

В заголовке `<string.h>` определен тип `size_t`; это тип результата, который получается после применения оператора¹ `sizeof` и представляет собой разновидность целого без знака.

В этой главе описаны только те функции, которые работают с символами типа `char`. Эти функции были определены стандартом С с самого начала, и, безусловно, они являются наиболее популярными и поддерживаются большинством компиляторов. Двухбайтовые функции, работающие с символами типа `wchar_t`, описаны в главе 19.

В версии С99 к некоторым параметрам нескольких функций, первоначально определенных в версии С89, добавлен квалификатор `restrict`. При рассмотрении каждой такой функции будет приведен ее прототип, используемый в среде С89 (а также в среде С++), а параметры с атрибутом `restrict` будут отмечены в описании этой функции.



Функция `isalnum`

```
#include <ctype.h>
int isalnum(int ch);
```

Если аргумент `ch` функции `isalnum()` является либо буквой, либо цифрой, она возвращает ненулевое значение. Если же тестируемый символ не относится к алфавитно-цифровым, возвращается нуль.

Пример

Данная программа читает из стандартного входного потока `stdin` символы, проверяет их и выдает сообщение о каждом алфавитно-цифровом символе.

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

int main(void)
{
```

¹ В данном случае под оператором подразумевается знак функции. — Прим. ред.


```

char ch;

for(;;) {
    ch = getc(stdin);
    if(ch == '.') break;
    if(isalnum(ch)) printf("Символ %с является алфавитно-цифровым\n",
ch);
}

return 0;
}

```

Зависимые функции

isalpha(), iscntrl(), isdigit(), isgraph(), isprint(), ispunct() и isspace()

Функция isalpha

```

#include <ctype.h>
int isalpha(int ch);

```

Функция `isalpha()` возвращает ненулевое значение, если ее аргумент `ch` является буквой, в противном случае возвращается нуль. Принадлежность символа к буквам зависит от конкретного языка. Для английского языка таковыми являются прописные и строчные буквы от A до Z.

Пример

Данная программа делает проверку каждого символа, прочитанного из стандартного входного потока `stdin`, и выдает сообщение, если этот символ окажется буквой.

```

#include <ctype.h>
#include <stdio.h>

int main(void)
{
    char ch;

    for(;;) {
        ch = getchar();
        if(ch == '.') break;
        if(isalpha(ch)) printf("%с является буквой\n", ch);
    }

    return 0;
}

```

Зависимые функции

isalnum(), iscntrl(), isdigit(), isgraph(), isprint(), ispunct() и isspace()

Функция `isblank`

```
#include <ctype.h>
int isblank(int ch);
```

Функция `isblank()` добавлена в версии C99.

Она возвращает ненулевое значение, если ее аргумент *ch* является символом, для которого функция `isspace()` возвращает значение “истина”. Этот символ используется в качестве разделителя слов. Так, для английского языка пробельными символами являются пробел и символ горизонтальной табуляции.

Пример

Данная программа проверяет все символы, прочитанные из стандартного входного потока `stdin`, и выдает сообщение о каждом пробельном символе.

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

int main(void)
{
    char ch;

    for(;;) {
        ch = getchar();
        if(ch == '.') break;
        if(isblank(ch)) printf("%c является разделителем слов\n", ch);
    }

    return 0;
}
```

Зависимые функции

`isalnum()`, `isalpha()`, `isctrl()`, `isdigit()`, `isgraph()`, `ispunct()` и `isspace()`

Функция `isctrl`

```
#include <ctype.h>
int isctrl(int ch);
```

Функция `isctrl()` возвращает ненулевое значение, если ее аргумент *ch* является управляющим символом, значение которого в ASCII-средах лежит в диапазоне между нулем и `0x1F` или равно `0x7F` (символ `DEL`). В противном случае возвращается нуль.

Пример

Данная программа проверяет все символы, прочитанные из стандартного входного потока `stdin`, и выдает сообщение о каждом управляющем символе.

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

int main(void)
```

```

{
    char ch;

    for(;;) {
        ch = getchar();
        if(ch == '.') break;
        if(iscntrl(ch)) printf("%c является управляющим символом\n", ch);
    }

    return 0;
}

```

Зависимые функции

isalnum(), isalpha(), isdigit(), isgraph(), isprint(), ispunct() и isspace()

Функция isdigit

```

#include <ctype.h>
int isdigit(int ch);

```

Функция `isdigit()` возвращает ненулевое значение, если ее аргумент *ch* является цифрой, т.е. попадает в диапазон 0-9. В противном случае возвращается нуль.

Пример

Данная программа проверяет каждый символ, прочитанный из стандартного входного потока `stdin`, и выдает сообщение, если этот символ окажется цифрой.

```

#include <ctype.h>
#include <stdio.h>

int main(void)
{
    char ch;

    for(;;) {
        ch = getchar();
        if(ch == '.') break;
        if(isdigit(ch)) printf("%c является цифрой\n", ch);
    }

    return 0;
}

```

Зависимые функции

isalnum(), isalpha(), iscntrl(), isgraph(), isprint(), ispunct() и isspace()

Функция isgraph

```
#include <ctype.h>
int isgraph(int ch);
```

Функция `isgraph()` возвращает ненулевое значение, если ее аргумент *ch* является любым печатаемым символом, но не пробелом. В противном случае возвращается нуль. Для ASCII-сред значения печатаемых символов лежат в диапазоне от 0x21 до 0x7E.

Пример

Данная программа проверяет все символы, прочитанные из стандартного входного потока `stdin`, и выдает сообщение о каждом печатаемом символе.

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

int main(void)
{
    char ch;

    for(;;) {
        ch = getchar();
        if(isgraph(ch)) printf("%c является печатаемым символом\n", ch);
        if(ch == '.') break;
    }

    return 0;
}
```

Зависимые функции

`isalnum()`, `isalpha()`, `isctrl()`, `isdigit()`, `isprint()`, `ispunct()` и `isspace()`

Функция islower

```
#include <ctype.h>
int islower(int ch);
```

Функция `islower()` возвращает ненулевое значение, если аргумент *ch* является строчной буквой. В противном случае возвращается нуль.

Пример

Данная программа проверяет все символы, прочитанные из стандартного входного потока `stdin`, и выдает сообщение о каждой строчной букве.

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

int main(void)
{
    char ch;
```

```

    for(;;) {
        ch = getchar();
        if(ch == '.') break;
        if(islower(ch)) printf("%c является строчной буквой\n", ch);
    }

    return 0;
}

```

Зависимые функции

isuper()

Функция isprint

```

#include <ctype.h>
int isprint(int ch);

```

Функция `isprint()` возвращает ненулевое значение, если аргумент *ch* является печатаемым символом, включая пробел. В противном случае возвращается нуль. В ASCII-средах значения печатаемых символов лежат в диапазоне от 0x20 до 0x7E.

Пример

Данная программа проверяет все символы, прочитанные из стандартного входного потока `stdin`, и выдает сообщение о каждом печатаемом символе.

```

#include <ctype.h>
#include <stdio.h>

int main(void)
{
    char ch;

    for(;;) {
        ch = getchar();
        if(islower(ch)) printf("символ %c является печатаемым\n", ch);
        if(ch == '.') break;
    }

    return 0;
}

```

Зависимые функции

isalnum(), isalpha(), iscntrl(), isdigit(), isgraph(), ispunct() и isspace()

Функция ispunct

```

#include <ctype.h>
int ispunct(int ch);

```

Функция `ispunct()` возвращает ненулевое значение, если аргумент *ch* является знаком пунктуации. В противном случае возвращается нуль. Под знаками пунктуации

подразумеваются все печатаемые символы за исключением пробела, которые не относятся к алфавитно-цифровым.

Пример

Данная программа проверяет все символы, прочитанные из стандартного входного потока `stdin`, и выдает сообщение о каждом знаке пунктуации.

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

int main(void)
{
    char ch;

    for(;;) {
        ch = getchar();
        if(ispunct(ch)) printf("%c является знаком пунктуации\n", ch);
        if(ch == '.') break;
    }

    return 0;
}
```

Зависимые функции

`isalnum()`, `isalpha()`, `iscntrl()`, `isdigit()`, `isgraph()` и `isspace()`



Функция `isspace`

```
#include <ctype.h>
int isspace(int ch);
```

Функция `isspace()` возвращает ненулевое значение, если аргумент `ch` является пробельным символом. (К пробельным символам, помимо пробела, относятся символы горизонтальной и вертикальной табуляции, перевода строки, возврата каретки и новой строки¹.) В противном случае возвращается нуль.

Пример

Данная программа проверяет все символы, прочитанные из стандартного входного потока `stdin`, и выдает сообщение о каждом пробельном символе.

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

int main(void)
{
    char ch;

    for(;;) {
        ch = getchar();
        if(isspace(ch)) printf("%c является пробельным символом\n", ch);
        if(ch == '.') break;
    }
}
```

¹ При локализации к этому списку могут быть добавлены и другие символы. — *Прим. ред.*

```
}  
  
return 0;  
}
```

Зависимые функции

isalnum(), isalpha(), isblank(), iscntrl(), isdigit(), isgraph() и ispunct()

■ Функция isupper

```
#include <ctype.h>  
int isupper(int ch);
```

Функция `isupper()` возвращает ненулевое значение, если аргумент *ch* является прописной буквой. В противном случае возвращается нуль.

Пример

Данная программа проверяет все символы, прочитанные из стандартного входного потока `stdin`, и выдает сообщение о каждой прописной букве.

```
#include <ctype.h>  
#include <stdio.h>  
  
int main(void)  
{  
    char ch;  
  
    for(;;) {  
        ch = getchar();  
        if(ch == '.') break;  
        if(isupper(ch)) printf("%c является прописной буквой\n", ch);  
    }  
  
    return 0;  
}
```

Зависимые функции

islower()

■ Функция isxdigit

```
#include <ctype.h>  
int isxdigit(int ch);
```

Функция `isxdigit()` возвращает ненулевое значение, если аргумент *ch* является шестнадцатеричной цифрой. В противном случае возвращается нуль. Шестнадцатеричная цифра должна попадать в один из следующих диапазонов: A-F, a-f или 0-9.

Пример

Данная программа проверяет все символы, прочитанные из стандартного входного потока `stdin`, и выдает сообщение о каждой шестнадцатеричной цифре.

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

int main(void)
{
    char ch;

    for(;;) {
        ch = getchar();
        if(ch == '.') break;
        if(isxdigit(ch)) printf("%с является шестнадцатеричной
                                цифрой\n", ch);
    }

    return 0;
}
```

Зависимые функции

`isalnum()`, `isalpha()`, `isctrl()`, `isdigit()`, `isgraph()`, `ispunct()` и `isspace()`

Функция `memchr`

```
#include <string.h>
void *memchr(const void *buffer, int ch, size_t count);
```

Функция `memchr()` просматривает массив, адресуемый параметром *buffer*, чтобы отыскать первое вхождение символа *ch* в первых *count* символах.

Эта функция возвращает указатель на первый из символов *ch*, входящих в массив *buffer*, или нулевой указатель, если символ *ch* не найден.

Пример

Данная программа выводит на экран сообщение из примера.

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

int main(void)
{
    char *p;

    p = memchr("строка из примера", ' ', 17);
    printf(p);

    return 0;
}
```


Еще один пример

Некоторые примеры, приведенные в качестве иллюстрации применения строковых функций, иногда носят “несколько учебный” характер и не всегда могут быть рекомендованы для профессионального программирования. Например, указание константы 17 в операторе `p = memchr("строка из примера", ' ', 17);` предыдущего примера едва ли может служить хорошим примером для программиста. (Кстати сказать, против подобного употребления констант автор предупреждал читателей в предыдущих главах. Просто в данной программе автор не хотел “затенять” основную идею посторонними деталями.) Подумайте, что будет, если изменить строку? Опять подсчитывать количество символов? Чтобы избежать этого, во многих руководствах предлагается использовать для этой цели функцию `strlen`. Вот какой пример применения приводится, например, в документации по Borland C++ Builder 5:

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

int main(void)
{
    char str[17];
    char *ptr;

    strcpy(str, "This is a string");
    ptr = (char *) memchr(str, 'r', strlen(str));
    if (ptr)
        printf("The character 'r' is at position: %d\n", ptr - str);
    else
        printf("The character was not found\n");
    return 0;
}
```

В данном примере, правда, константа указана в объявлении массива `str`. Однако этого можно было легко избежать, проинициализировав массив в объявлении. Правда, остается еще один недостаток: некоторое снижение эффективности кода из-за вызова функции `strlen`. Но и его можно устранить, если длина строки может быть вычислена при компиляции (как в наших примерах). Действительно, тогда ведь можно воспользоваться операцией `sizeof` и записать не константу 17, а константное выражение `sizeof(str)/sizeof(char)`. (Отсюда вывод: никогда не верьте тем, кто говорит, что последовательное применение технологии программирования приводит к ухудшению характеристик программы. Если язык программирования хорошо сконструирован, то ничего подобного быть не может. Правда, если язык продуман плохо... то едва ли стоит на нем писать программы!)¹

Зависимые функции

`memcpy()` и `isspace()`

Функция `memcpy`

```
#include <string.h>
int memcpy(const void *buf1, const void *buf2, size_t count);
```

¹ Этот раздел добавлен редактором перевода. — *Прим. ред.*

Функция `memcmp()` сравнивает первые *count* символов массивов, адресуемых параметрами *buf1* и *buf2*.

Функция `memcmp()` возвращает целое значение, которое интерпретируется следующим образом.

Значение	Результат сравнения
Меньше нуля	<i>buf1</i> меньше <i>buf2</i>
Нуль	<i>buf1</i> равен <i>buf2</i>
Больше нуля	<i>buf1</i> больше <i>buf2</i>

Пример

Данная программа выдает результат сравнения двух своих аргументов, которые задаются в командной строке.

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

int main(int argc, char *argv[])
{
    int outcome, len, l1, l2;

    if(argc!=3) {
        printf("Неверно задано число аргументов.");
        exit(1);
    }

    /* определение длины более короткой строки */
    l1 = strlen(argv[1]);
    l2 = strlen(argv[2]);
    len = l1 < l2 ? l1:l2;
    outcome = memcmp(argv[1], argv[2], len);
    if(!outcome) printf("Равны");
    else if(outcome<0) printf("Первый меньше второго.");
    else printf("Первый больше второго.");

    return 0;
}
```

Зависимые функции

`memchr()`, `memcpy()` и `strncpy()`

Функция `memcpy`

```
#include <string.h>
void *memcpy(void *to, const void *from, size_t count);
```

Функция `memcpy()` копирует *count* символов из массива, адресуемого параметром *from*, в массив, адресуемый параметром *to*. Если заданные массивы перекрываются, поведение функции `memcpy()` не определено.

В версии C99 к параметрам *to* и *from* применен квалификатор `restrict`.

Функция `memcpy()` возвращает значение указателя *to*.

Пример

Данная программа копирует содержимое массива `buf1` в массив `buf2` и выводит результат на дисплей.

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

#define SIZE 80

int main(void)
{
    char buf1[SIZE], buf2[SIZE];

    strcpy(buf1, "Когда, в случае если...");
    memcpy(buf2, buf1, SIZE);
    printf(buf2);

    return 0;
}
```

Зависимые функции

`memcpy()`

Функция `memmove`

```
#include <string.h>
void *memmove(void *to, const void *from, size_t count);
```

Функция `memmove()` копирует *count* символов из массива, адресуемого параметром *from*, в массив, адресуемый параметром *to*. Если заданные массивы перекрываются, процесс копирования проходит корректно, т.е. соответствующее содержимое будет помещено в массив *to*, но содержимое массива *from* при этом изменится.

Функция `memmove()` возвращает значение указателя *to*.

Пример

Данная программа сдвигает содержимое массива `str` на 10 позиций в сторону младших адресов и выводит результат на дисплей.

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

#define SIZE 80

int main(void)
{
    char str[SIZE], *p;

    strcpy(str, "Когда, в случае если...");
    p = str + 10;

    memmove(str, p, SIZE);
    printf("результат сдвига: %s", str);

    return 0;
}
```

Зависимые функции

memcpy()

Функция memset

```
#include <string.h>
void *memset(void *buf, int ch, size_t count);
```

Функция `memset()` копирует младший байт параметра *ch* в первые *count* символов массива, адресуемого параметром *buf*. Функция возвращает значение указателя *buf*.

Чаще всего функция `memset()` используется для инициализации области памяти некоторым известным значением.

Пример

Данный фрагмент инициализирует первые 100 байтов массива, адресуемого указателем *buf*, нулями. Затем он помещает символы X в первые 10 байтов этого массива и выводит строку XXXXXXXXXXXX.

```
memset(buf, '\0', 100);
memset(buf, 'X', 10);
printf(buf);
```

Зависимые функции

memcpy(), memset() и memmove()

Функция strcat

```
#include <string.h>
char *strcat(char *str1, const char *str2);
```

Функция `strcat()` присоединяет к строке *str1* копию строки *str2* и завершает строку *str1* нулевым символом. Конечный нуль-символ, первоначально завершающий строку *str1*, перезаписывается первым символом строки *str2*. Строка *str2* при этом не изменяется. Если заданные массивы перекрываются, поведение функции `strcat()` не определено.

В версии C99 к параметрам *str1* и *str2* применен квалификатор `restrict`.

Функция `strcat()` возвращает значение указателя *str1*.

Помните, что при выполнении операций с массивами символов контроль нарушения их границ не выполняется, поэтому программист должен сам позаботиться о достаточном размере массива *str1*, позволяющем вместить как его исходное содержимое, так и содержимое массива *str2*.

Пример

Данная программа дописывает первую строку, прочитанную из стандартного входного потока `stdin`, ко второй строке. Например, если пользователь введет строки привет и всем, то программа выведет сообщение всемпривет.

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

```
int main(void)
{
    char s1[80], s2[80];

    gets(s1);
    gets(s2);

    strcat(s2, s1);
    printf(s2);

    return 0;
}
```

Зависимые функции

`strchr()`, `strcmp()` и `strcpy()`



Функция `strchr`

```
#include <string.h>
char *strchr(const char *str, int ch);
```

Функция `strchr()` возвращает указатель на первое вхождение младшего байта параметра *ch* в строку *str*. Если указанный символ не найден, возвращается нулевой указатель.

Пример

Данная программа выводит строку из примера.

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

int main(void)
{
    char *p;

    p = strchr("строка из примера", ' ');
    printf(p);

    return 0;
}
```

Зависимые функции

`strpbrk()`, `strspn()`, `strstr()` и `strtok()`



Функция `strcmp`

```
#include <string.h>
int strcmp(const char *str1, const char *str2);
```

Функция `strcmp()` сравнивает в лексикографическом порядке две строки и возвращает целое значение, зависящее следующим образом от результата сравнения.

Значение	Результат сравнения строк
Меньше нуля	<i>str1</i> меньше <i>str2</i>
Ноль	<i>str1</i> равна <i>str2</i>
Больше нуля	<i>str1</i> больше <i>str2</i>

Пример

Следующую функцию можно использовать для проверки пароля. В случае неудачи она возвращает ноль, а при успешном выполнении — единицу.

```
int password(void)
{
    char s[80];

    printf("Введите пароль: ");
    gets(s);

    if(strcmp(s, "пароль")) {
        printf("Неверный пароль\n");
        return 0;
    }
    return 1;
}
```

Зависимые функции

`strchr()`, `strcpy()` и `strcmp()`

Функция strcoll

```
#include <string.h>
int strcoll(const char *str1, const char *str2);
```

Функция `strcoll()` сравнивает строку, адресуемую указателем *str1*, со строкой, адресуемой указателем *str2*. Сравнение выполняется с учетом значения параметра *locale*, заданного с помощью функции `setlocale()` (подробности приводятся в описании функции `setlocale()`).

Функция `strcoll()` возвращает целое значение, которое интерпретируется следующим образом.

Значение	Результат сравнения
Меньше нуля	<i>str1</i> меньше <i>str2</i>
Ноль	<i>str1</i> равно <i>str2</i>
Больше нуля	<i>str1</i> больше <i>str2</i>

Пример

Данный фрагмент программы выводит на экран сообщение Равно.

```
if(strcoll("привет", "привет")) printf("Равно");
```

Зависимые функции

`memcmp()` и `strcmp()`

Функция strcpy

```
#include <string.h>
char *strcpy(char *str1, const char *str2);
```

Функция `strcpy()` копирует содержимое строки *str2* в строку *str1*. Параметр *str2* должен указывать на строку с завершающим нулевым символом. Функция `strcpy()` возвращает значение указателя *str1*.

В версии C99 к параметрам *str1* и *str2* применен квалификатор `restrict`.

Если символьные массивы *str1* и *str2* перекрываются, поведение функции `strcpy()` не определено.

Пример

Следующий фрагмент программы копирует строку привет в строку *str*.

```
char str[80];
strcpy(str, "привет");
```

Зависимые функции

`memcpy()`, `strchr()`, `strcmp()` и `strncmp()`

Функция strcspn

```
#include <string.h>
size_t strcspn(const char *str1, const char *str2);
```

Функция `strcspn()` возвращает длину начальной подстроки в строке, адресуемой параметром *str1*, которая не содержит ни одного символа из строки, адресуемой параметром *str2*. Другими словами, функция `strcspn()` возвращает индекс первого символа в строке *str1*, который совпадает с любым из символов в строке *str2*.

Пример

Следующая программа выводит число 6.

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

int main(void)
{
    int len;

    len = strcspn("это тест", "сл");
    printf("%d", len);

    return 0;
}
```

Зависимые функции

`strrchr()`, `strpbrk()`, `strstr()` и `strtok()`

Функция `strerror`

```
#include <string.h>
char *strerror(int errnum);
```

Функция `strerror()` возвращает указатель на строку, содержащую системное сообщение об ошибке, связанной со значением *errnum*. Эту строку не следует менять ни при каких обстоятельствах.

Пример

Данный фрагмент программы выводит на экран системное сообщение об ошибке.

```
printf(strerror(10));
```

Функция `strlen`

```
#include <string.h>
size_t strlen(const char *str);
```

Функция `strlen()` возвращает длину строки, адресуемой параметром *str*, причем строка должна заканчиваться символом конца строки. Символ конца строки (`'0'`) не учитывается.

Пример

Данный фрагмент программы выводит на экран число 6.

```
printf("%d", strlen("привет"));
```

Зависимые функции

`memcpy()`, `strchr()`, `strcmp()` и `strncmp()`

Функция `strncat`

```
#include <string.h>
char *strncat(char *str1, const char *str2, size_t count);
```

Функция `strncat()` присоединяет к строке, адресуемой параметром *str1*, не более *count* символов строки, адресуемой параметром *str2*, завершая “результатирующую” строку *str1* нулевым символом. Конечный нуль-символ, первоначально завершающий строку *str1*, перезаписывается первым символом строки *str2*. Строка *str2* в результате этой операции конкатенации не меняется. Если строки перекрываются, поведение функции `strncat()` не определено.

В версии C99 к параметрам *str1* и *str2* применен квалификатор `restrict`.

Функция `strncat()` возвращает значение указателя *str1*.

При выполнении операций с массивами символов контроль нарушения их границ не выполняется, поэтому программист должен сам позаботиться о достаточном размере массива *str1*, позволяющем вместить как его исходное содержимое, так и содержащее присоединяемого массива *str2*.

Пример

Данная программа конкатенирует первую строку, прочитанную из стандартного входного потока `stdin`, ко второй строке и предотвращает переполнение массива `s1`, в который записывается результат. Например, если пользователь введет сначала привет, а затем всем, то программа выведет сообщение всемпривет.

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

int main(void)
{
    char s1[80], s2[80];
    unsigned int len;

    gets(s1);
    gets(s2);

    /* вычисление подходящей длины */
    len = 79 - strlen(s2);

    strncat(s2, s1, len);
    printf(s2);

    return 0;
}
```

Зависимые функции

`strcat()`, `strchr()`, `strncmp()` и `strncpy()`

■ Функция `strncmp`

```
#include <string.h>
int strncmp(const char *str1, const char *str2, size_t count);
```

Функция `strncmp()` сравнивает в лексикографическом порядке не более *count* символов из двух строк, заканчивающихся символом конца строки, и возвращает целое значение, зависящее от результата сравнения следующим образом:

Значение	Результат сравнения строк
Меньше нуля	<i>str1</i> меньше <i>str2</i>
Нуль	<i>str1</i> равна <i>str2</i>
Больше нуля	<i>str1</i> больше <i>str2</i>

Если в какой-нибудь из заданных строк меньше *count* символов, сравнение заканчивается при обнаружении первого нулевого символа.

Пример

Следующая функция сравнивает первые восемь символов двух своих аргументов, взятых из командной строки, и выдаст сообщение в случае их равенства.

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

```

int main(int argc, char *argv[])
{
    if(argc!=3) {
        printf("Неверное количество аргументов.");
        exit(1);
    }

    if(!strncmp(argv[1], argv[2], 8))
        printf("Строки одинаковые.\n");

    return 0;
}

```

Зависимые функции

strcmp(), strchr() и strncpy()

Функция strncpy

```

#include <string.h>
char *strncpy(char *str1, const char *str2, size_t count);

```

Функция `strncpy()` копирует не более *count* символов из строки, адресуемой параметром *str2*, в массив, адресуемый параметром *str1*. Параметр *str2* должен указывать на строку, заканчивающуюся символом конца строки.

В версии C99 к параметрам *str1* и *str2* применен квалификатор `restrict`.

Если заданные массивы символов перекрываются, поведение функции `strncpy()` не определено.

Если длина строки, адресуемой параметром *str2*, меньше значения *count*, то в конец строки-результата *str1* добавляются “недостающие” нулевые символы.

Если же длина строки, адресуемой параметром *str2*, больше значения *count*, то строка-результат, адресуемая параметром *str1*, не будет заканчиваться символом конца строки¹.

Функция `strncpy()` возвращает значение указателя *str1*.

Пример

Следующий фрагмент программы копирует не более 79 символов из строки *str1* в строку *str2*, тем самым гарантируется, что массив не переполнится.

```

char str1[128], str2[80];

gets(str1);
strncpy(str2, str1, 79);

```

Зависимые функции

memcpy(), strchr(), strncat() и strncmp()

¹ Обратите внимание, что такую “строку” считать полноценной нельзя, ведь некоторые строковые функции работать с подобными строками могут некорректно. — Прим. ред.

■ Функция `strpbrk`

```
#include <string.h>
char *strpbrk(const char *str1, const char *str2);
```

Функция `strpbrk()` возвращает указатель на первый символ в строке, адресуемой параметром *str1*, который совпадает с любым символом в строке, адресуемой параметром *str2*. Символы конца строки, которыми должны оканчиваться данные строки, в расчет не берутся. Если совпадений нет, возвращается нулевой указатель.

Пример

Данная программа выводит на экран сообщение о тест.

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

int main(void)
{
    char *p;

    p = strpbrk("это тест", " есою");
    printf(p);

    return 0;
}
```

Зависимые функции

`strspn()`, `strrchr()`, `strstr()` и `strtok()`

■ Функция `strrchr`

```
#include <string.h>
char *strrchr(const char *str, int ch);
```

Функция `strrchr()` возвращает указатель на последнее вхождение младшего байта параметра *ch* в строку, адресуемую параметром *str*. Если совпадение не обнаружено, возвращается нулевой указатель.

Пример

Данная программа выводит сообщение пыль летит.

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

int main(void)
{
    char *p;

    p = strrchr("по полю пыль летит", 'п');
    printf(p);

    return 0;
}
```

Зависимые функции

strpbrk(), strspn(), strstr() и strtok()

Функция strspn

```
#include <string.h>
size_t strspn(const char *str1, const char *str2);
```

Функция `strspn()` возвращает длину начальной подстроки строки, адресуемой параметром `str1`, которая состоит только из символов, содержащихся в строке, адресуемой параметром `str2`. Другими словами, функция `strspn()` возвращает индекс первого символа в строке `str1`, который не совпадает ни с одним из символов в строке `str2`¹.

Пример

Эта программа выводит число 11.

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

int main(void)
{
    int len;

    len=strspn("это строка из примера", "акортэ с");
    printf("%d", len);

    return 0;
}
```

Зависимые функции

strpbrk(), strrchr(), strstr() и strtok()

Функция strstr

```
#include <string.h>
char *strstr(const char *str1, const char *str2);
```

Функция `strstr()` возвращает указатель на первое вхождение подстроки, адресуемой параметром `str2`, в строку, адресуемую параметром `str1`. Если совпадение не обнаружено, возвращается нулевой указатель.

Пример

Данная программа выводит сообщение от неуправления.

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

¹ Или (что то же самое) функция `strspn()` возвращает индекс первого символа в строке `str1`, который не входит в строку `str2`. — *Прим. ред.*

```
int main(void)
{
    char *p;

    p = strstr("хлопот невпроворот", "от");
    printf(p);

    return 0;
}
```

Зависимые функции

strchr(), strcspn(), strpbrk(), strspn(), strtok() и strrchr()

■ Функция strtok

```
#include <string.h>
char *strtok(char *str1, const char *str2);
```

Функция `strtok()` возвращает указатель на следующую лексему в строке, адресуемой параметром `str1`. Символы, образующие строку, адресуемую параметром `str2`, представляют собой разделители, которые определяют лексему. При отсутствии лексемы, подлежащей возврату, возвращается нулевой указатель.

В версии C99 к параметрам `str1` и `str2` применен квалификатор `restrict`.

Чтобы разделить некоторую строку на лексемы, при первом вызове функции `strtok()` параметр `str1` должен указывать на начало этой строки. При последующих вызовах функции в качестве параметра `str1` нужно использовать нулевой указатель. Этим способом вся строка разбивается на лексемы.

При каждом обращении к функции `strtok()` можно использовать различные наборы разделителей.

Пример

Эта программа разбивает строку "Травка зеленеет, солнышко блестит" на лексемы, разделителями которых служат пробелы и запятые. В результате получится

Травка|зеленеет|солнышко|блестит

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

int main(void)
{
    char *p;

    p = strtok("Травка зеленеет, солнышко блестит", " ");
    printf(p);
    do {
        p = strtok('\0', ", ");
        if(p) printf("|%s", p);
    } while(p);

    return 0;
}
```

Зависимые функции

`strchr()`, `strcspn()`, `strpbrk()`, `strrchr()` и `strspn()`

Функция `strxfrm`

```
#include <string.h>
size_t strxfrm(char *str1, const char *str2, size_t count);
```

Функция `strxfrm()` преобразует строку, адресуемую параметром *str2*, таким образом, чтобы ее можно было использовать функцией `strcmp()`, и помещает результат преобразования в строку, адресуемую параметром *str1*. После преобразования результат вызова функции `strcmp()`, использующей параметр *str1*, будет совпадать с результатом вызова функции `strcoll()`, использующей исходную строку, на которую указывает параметр *str2*. В массив, адресуемый параметром *str1*, записывается не более *count* символов.

В версии C99 к параметрам *str1* и *str2* применен квалификатор `restrict`.

Функция `strxfrm()` возвращает длину преобразованной строки¹.

Пример

Данный фрагмент программы (одна строка) преобразует первые 10 символов строки, адресуемой указателем *s2*, и помещает результат преобразования в строку, адресуемую указателем *s1*.

```
strxfrm(s1, s2, 10);
```

Зависимые функции

`strcoll()`

Функция `tolower`

```
#include <ctype.h>
int tolower(int ch);
```

Функция `tolower()` возвращает строчный эквивалент параметра *ch*, если он является буквой; в противном случае возвращается *ch* без изменений.

Пример

Этот фрагмент программы выводит букву *a*.

```
putchar(tolower('A'));
```

¹ В связи с этим может возникнуть вопрос: что это за таинственное преобразование, которому подвергается строка *str2*, и зачем оно нужно? Действительно, в списке зависимых функций автор не приводит каких-либо функций, задающих такое преобразование, или способных влиять на него. Все дело в том, что это преобразование предназначено для строк, выводимых локализованными программами, в которых учитывается местная специфика, т.е. национальная и культурная среда, в которой функционирует система или программа. Установка местной спецификации может выполняться геополитической функцией `setlocale()`. Ее вызов в случае, например, кодовой страницы 850 для французского языка может выглядеть так: `setlocale(LC_ALL, "French_France.850")`; — *Прим. ред.*

Зависимые функции

`toupper()`

Функция `toupper`

```
#include <ctype.h>
int toupper(int ch);
```

Функция `toupper()` возвращает прописной эквивалент параметра *ch*, если *ch* — буква; в противном случае *ch* возвращается без изменений.

Пример

Этот фрагмент программы выводит букву А.

```
putchar(toupper('a'));
```

Зависимые функции

`tolower()`

Полный
справочник по



Глава 15

Математические функции

В версии C99 математическая библиотека была значительно пополнена; при этом число ее функций увеличилось более чем в три раза (стандарт C89 определял всего лишь 22 математические функции). Одной из основных целей комитета по версии C99 было повышение применимости языка C для численных расчетов. Теперь с уверенностью можно сказать, что эта цель достигнута!

Для использования математических функций в программу необходимо включить заголовок `<math.h>`. Помимо объявления математических функций, этот заголовок определяет один или несколько макросов. В версии C89 заголовком `<math.h>` определяется только макрос `HUGE_VAL`, который представляет собой значение типа `double`, сигнализирующее о возникшем переполнении. В версии C99 кроме него определены следующие макросы.

<code>HUGE_VALF</code>	версия макроса <code>HUGE_VAL</code> с типом <code>float</code>
<code>HUGE_VALL</code>	версия макроса <code>HUGE_VAL</code> с типом <code>long double</code>
<code>INFINITY</code>	Значение, представляющее бесконечность
<code>math_errhandling</code>	Содержит макросы <code>MATH_ERRNO</code> и/или <code>MATH_ERREXCEPT</code>
<code>MATH_ERRNO</code>	Встроенная глобальная переменная <code>errno</code> , используемая для вывода сообщений об ошибках
<code>MATH_ERREXCEPT</code>	Исключение, возбуждаемое при выполнении операций над вещественными числами, с целью вывода сообщения об ошибках
<code>NAN</code>	Не число

В версии C99 определены следующие макросы (подобные функциям), классифицирующие значение.

<code>int fpclassify(<i>fpval</i>)</code>	В зависимости от значения аргумента <i>fpval</i> возвращает <code>FP_INFINITY</code> , <code>FP_NAN</code> , <code>FP_NORMAL</code> , <code>FP_SUBNORMAL</code> или <code>FP_ZERO</code> . Эти макросы определяются заголовком <code><math.h></code>
<code>int isfinite(<i>fpval</i>)</code>	Возвращает ненулевое значение, если <i>fpval</i> конечное
<code>int isinf(<i>fpval</i>)</code>	Возвращает ненулевое значение, если <i>fpval</i> представляет бесконечность
<code>int isnan(<i>fpval</i>)</code>	Возвращает ненулевое значение, если <i>fpval</i> — не является числом
<code>int isnormal(<i>fpval</i>)</code>	Возвращает ненулевое значение, если <i>fpval</i> представляет собой нормализованное число
<code>int signbit(<i>fpval</i>)</code>	Возвращает ненулевое значение, если <i>fpval</i> отрицательно (т.е. установлен его знаковый разряд)

В версии C99 определены следующие макросы сравнения, аргументы которых (*a* и *b*) должны иметь числовые значения в формате с плавающей точкой.

<code>int isgreater(<i>a</i>, <i>b</i>)</code>	Возвращает ненулевое значение, если <i>a</i> больше <i>b</i>
<code>int isgreaterequal(<i>a</i>, <i>b</i>)</code>	Возвращает ненулевое значение, если <i>a</i> больше или равно <i>b</i>
<code>int isless(<i>a</i>, <i>b</i>)</code>	Возвращает ненулевое значение, если <i>a</i> меньше <i>b</i>
<code>int islessequal(<i>a</i>, <i>b</i>)</code>	Возвращает ненулевое значение, если <i>a</i> меньше или равно <i>b</i>
<code>int islessgreater(<i>a</i>, <i>b</i>)</code>	Возвращает ненулевое значение, если <i>a</i> больше или меньше <i>b</i>
<code>int isunordered(<i>a</i>, <i>b</i>)</code>	Возвращает 1, если <i>a</i> и <i>b</i> не упорядочены одно по отношению к другому Возвращает 0, если <i>a</i> и <i>b</i> упорядочены

Эти макросы введены, так как они прекрасно обрабатывают значения, которые не являются числами, не вызывая при этом исключений вещественного типа.

Макросы `EDOM` и `ERANGE` также используются математическими функциями. Эти макросы определены в заголовке `<errno.h>`.

Ошибки в версиях C89 и C99 обрабатываются по-разному. Так, в версии C89, если аргумент математической функции не попадает в область определения, возвращается некоторое значение, зависящее от конкретной реализации, а встроенная глобальная

целая переменная `errno` устанавливается равной значению `EDOM`. В версии C99 нарушение области определения также приводит к возврату значения, зависящего от конкретной реализации. Однако по значению `math_errhandling` можно судить о выполнении других действий. Если `math_errhandling` содержит значение `MATH_ERRNO`, то встроенная глобальная целая переменная `errno` устанавливается равной значению `EDOM`. Если же `math_errhandling` содержит значение `MATH_ERREXCEPT`, возбуждается исключение вещественного типа.

В версии C89, если функция генерирует результат, который слишком велик и потому не может быть представлен в машинном формате, то происходит переполнение. В этом случае функция возвращает значение `HUGE_VAL`, а переменная `errno` устанавливается равной значению `ERANGE`, сигнализирующему о выходе за пределы диапазона. При потере значимости функция возвращает нуль и устанавливает переменную `errno` равной значению `ERANGE`. В версии C99 ошибка переполнения также приводит к тому, что функция возвращает значение `HUGE_VAL`, а при потере значимости — нуль. Если `math_errhandling` содержит значение `MATH_ERRNO`, глобальная переменная `errno` устанавливается равной значению `ERANGE`, свидетельствующему об ошибке диапазона. Если же `math_errhandling` содержит значение `MATH_ERREXCEPT`, возбуждается исключение вещественного типа.

В версии C89 аргументами математических функций должны быть значения типа `double`, причем значения, возвращаемые функциями, тоже имеют тип `double`. В версии C99 добавлены варианты этих функций, работающие с типами `float` и `long double`. В этих функциях используются суффиксы `f` и `l` соответственно. Например, в версии C89 функция `sin()` определена следующим образом.

```
double sin(double arg);
```

Версия C99 поддерживает приведенное выше определение функции `sin()`, но в ней добавлены еще две ее модификации — `sinf()` и `sinl()`.

```
float sinf(float arg);  
long double sinl(long double arg);
```

Операции, выполняемые всеми тремя функциями, одинаковы; различаются лишь типы данных, над которыми выполняются эти операции. Добавление модификаций `f` и `l` математических функций позволяет использовать версию, которая наиболее точно соответствует данным, с которыми работают функции.

Поскольку в версии C99 добавлено так много новых функций, стоит отдельно перечислить те из них, которые поддерживаются версией C89. Это следующие функции.

<code>acos</code>	<code>cos</code>	<code>fmod</code>	<code>modf</code>	<code>tan</code>
<code>asin</code>	<code>cosh</code>	<code>frexp</code>	<code>pow</code>	<code>tanh</code>
<code>atan</code>	<code>exp</code>	<code>ldexp</code>	<code>sin</code>	
<code>atan2</code>	<code>fabs</code>	<code>log</code>	<code>sinh</code>	
<code>ceil</code>	<code>floor</code>	<code>log10</code>	<code>sqrt</code>	

И последнее замечание: все углы задаются в радианах.

Семейство функций `acos`

```
#include <math.h>  
float acosf(float arg);  
double acos(double arg);  
long double acosl(long double arg);
```

Функции `acosf()` и `acosl()` добавлены в версии C99.

Каждая функция семейства `acos()` возвращает значение арккосинуса от аргумента *arg*. Значение аргумента должно находиться в диапазоне от -1 до 1; в противном случае возникает ошибка из-за выхода за пределы области допустимых значений (ошибка из-за нарушения области определения).

Пример

Данная программа выводит значения арккосинусов последовательности аргументов, лежащих в интервале от -1 до 1 и увеличивающихся с шагом одна десятая, т.е. программа составляет таблицу арккосинуса.

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

int main(void)
{
    double val = -1.0;

    do {
        printf("Арккосинус %f равен %f.\n", val, acos(val));
        val += 0.1;
    } while(val<=1.0);

    return 0;
}
```

Зависимые функции

`asin()`, `atan()`, `atan2()`, `sin()`, `cos()`, `tan()`, `sinh()`, `cosh()` и `tanh()`

Семейство функций `acosh`

```
#include <math.h>
float acoshf(float arg);
double acosh(double arg);
long double acoshl(long double arg);
```

Функции `acosh()`, `acoshf()` и `acoshl()` добавлены в версии C99.

Каждая функция семейства `acosh()` возвращает значение гиперболического арккосинуса от аргумента *arg*. Значение аргумента должно быть больше или равно нулю; в противном случае возникает ошибка из-за выхода за пределы области допустимых значений (ошибка из-за нарушения области определения).

Зависимые функции

`asinh()`, `atanh()`, `sinh()`, `cosh()` и `tanh()`

Семейство функций `asin`

```
#include <math.h>
float asinf(float arg);
double asin(double arg);
long double asinl(long double arg);
```

Функции `asinf()` и `asinl()` добавлены в версии C99.

Каждая функция семейства `asin()` возвращает значение арксинуса от аргумента *arg*. Значение аргумента должно находиться в диапазоне от -1 до 1; в противном случае возникает ошибка из-за выхода за пределы области допустимых значений (ошибка из-за нарушения области определения).

Пример

Данная программа выводит значения арксинусов последовательности аргументов, лежащих в интервале от -1 до 1 и увеличивающихся с шагом одна десятая, т.е. составляет таблицу арксинуса.

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

int main(void)
{
    double val = -1.0;

    do {
        printf("Арксинус %f равен %f.\n", val, asin(val));
        val += 0.1;
    } while(val<=1.0);

    return 0;
}
```

Зависимые функции

`acos()`, `atan()`, `atan2()`, `sin()`, `cos()`, `tan()`, `sinh()`, `cosh()` и `tanh()`

Семейство функций `asinh`

```
#include <math.h>
float asinhf(float arg);
double asinh(double arg);
long double asinhl(long double arg);
```

Функции `asinh()`, `asinhf()` и `asinhl()` добавлены в версии C99.

Каждая функция семейства `asinh()` возвращает значение гиперболического арксинуса от аргумента *arg*.

Зависимые функции

`acosh()`, `atanh()`, `sinh()`, `cosh()` и `tanh()`

Семейство функций `atan`

```
#include <math.h>
float atanf(float arg);
double atan(double arg);
long double atanl(long double arg);
```

Функции `atanf()` и `atanl()` добавлены в версии C99.

Каждая функция семейства `atan()` возвращает значение арктангенса от аргумента *arg*.

Пример

Данная программа выводит значения арктангенсов последовательности аргументов, лежащих в интервале от -1 до 1 и увеличивающихся с шагом одна десятая, т.е. составляет таблицу арктангенса.

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

int main(void)
{
    double val = -1.0;

    do {
        printf("Арктангенс %f равен %f.\n", val, atan(val));
        val += 0.1;
    } while(val<=1.0);

    return 0;
}
```

Зависимые функции

asin(), acos(), atan2(), tan(), cos(), sin(), sinh(), cosh() и tanh()

Семейство функций atanh

```
#include <math.h>
float atanhf(float arg);
double atanh(double arg);
long double atanh1(long double arg);
```

Функции atanh(), atanhf() и atanh1() добавлены в версии C99.

Каждая функция семейства atanh() возвращает значение гиперболического арктангенса от аргумента *arg*. Значение аргумента должно находиться в диапазоне от -1 до 1 (не включая границы); в противном случае возникает ошибка из-за выхода за пределы области допустимых значений (ошибка из-за нарушения области определения). Если *arg* равен 1 или -1, возможен выход за пределы допустимого диапазона.

Зависимые функции

acosh(), asinh(), sinh(), cosh() и tanh()

Семейство функций atan2

```
#include <math.h>
float atan2f(float a, float b);
double atan2(double a, double b);
long double atan2l(long double a, long double b);
```

Функции atan2f() и atan2l() добавлены в версии C99.

Каждая функция семейства atan2() возвращает значение арктангенса отношения *a/b*. Для вычисления квадранта возвращаемого значения используются знаки аргументов функции.

Пример

Данная программа выводит значения арктангенсов последовательности аргументов y , лежащих в интервале от -1 до 1 и увеличивающихся с шагом одна десятая, т.е. составляет таблицу арктангенса.

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

int main(void)
{
    double val = -1.0;

    do {
        printf("Арктангенс %f равен %f.\n", val, atan2(val, 1.0));
        val += 0.1;
    } while(val<=1.0);

    return 0;
}
```

Зависимые функции

asin(), acos(), atan(), tan(), cos(), sin(), sinh(), cosh() и tanh()

■ Семейство функций cbrt

```
#include <math.h>
float cbrtf(float num);
double cbrt(double num);
long double cbrtl(long double num);
```

Функции cbrt(), cbrtf() и cbrtl() добавлены в версии C99.

Каждая функция семейства cbrt() возвращает значение кубического корня от аргумента *num*.

Пример

Данный фрагмент программы выводит на экран число 2.

```
printf("%f", cbrt(8));
```

Зависимые функции

sqrt()

■ Семейство функций ceil

```
#include <math.h>
float ceilf(float num);
double ceil(double num);
long double ceill(long double num);
```

Функции ceilf() и ceill() добавлены в версии C99.

Каждая функция семейства `ceil()` возвращает наименьшее целое (представленное в виде значения с плавающей точкой), которое больше значения аргумента *num* или равно ему. Например, если *num* равно 1.02, функция `ceil()` вернет значение 2.0, а при *num*, равном -1.02, — значение -1.

Пример

Данный фрагмент программы выводит на экран число 10.

```
printf("%f", ceil(9.9));
```

Зависимые функции

`floor()` и `fmod()`

Семейство функций `copysign`

```
#include <math.h>
float copysignf(float val, float signval);
double copysign(double val, double signval);
long double copysignl(long double val, long double signval);
```

Функции `copysign()`, `copysignf()` и `copysignl()` добавлены в версии C99.

Каждая функция семейства `copysign()` наделяет аргумент *val* знаком, который имеет аргумент *signval*, и возвращает полученный результат. Таким образом, возвращаемое значение имеет величину, равную величине аргумента *val*, а его знак совпадает со знаком аргумента *signval*.

Зависимые функции

`fabs()`

Семейство функций `cos`

```
#include <math.h>
float cosf(float arg);
double cos(double arg);
long double cosl(long double arg);
```

Функции `cosf()` и `cosl()` добавлены в версии C99.

Каждая функция семейства `cos()` возвращает значение косинуса аргумента *arg*. Значение аргумента должно быть выражено в радианах.

Пример

Данная программа выводит значения косинусов последовательности аргументов, лежащих в интервале от -1 до 1 и увеличивающихся с шагом одна десятая, т.е. составляет таблицу косинуса.

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

int main(void)
{
```

```
double val = -1.0;

do {
    printf("Косинус %f равен %f.\n", val, cos(val));
    val += 0.1;
} while(val<=1.0);

return 0;
}
```

Зависимые функции

asin(), acos(), atan2(), atan(), tan(), sin(), sinh(), cos() и tanh()

■ Семейство функций cosh

```
#include <math.h>
float coshf(float arg);
double cosh(double arg);
long double coshl(long double arg);
```

Функции coshf() и coshl() добавлены в версии C99.

Каждая функция семейства cosh() возвращает значение гиперболического косинуса аргумента *arg*.

Пример

Данная программа выводит значения гиперболических косинусов последовательности аргументов, лежащих в интервале от -1 до 1 и увеличивающихся с шагом одна десятая, т.е. составляет таблицу гиперболического косинуса.

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

int main(void)
{
    double val = -1.0;

    do {
        printf("Гиперболический косинус %f равен %f.\n", val, cosh(val));
        val += 0.1;
    } while(val<=1.0);

    return 0;
}
```

Зависимые функции

asin(), acos(), atan2(), atan(), tan(), sin() и tanh()

■ Семейство функций erf

```
#include <math.h>
float erff(float arg);
```



```
double erf(double arg);
long double erfl(long double arg);
```

Функции `erf()`, `erff()` и `erfl()` добавлены в версии C99.

Каждая функция семейства `erf()` возвращает значение функции ошибок¹ от аргумента *arg*.

Зависимые функции

`erfc()`

Семейство функций `erfc`

```
#include <math.h>
float erfcf(float arg);
double erfc(double arg);
long double erfcl(long double arg);
```

Функции `erfc()`, `erfcf()` и `erfcl()` добавлены в версии C99.

Каждая функция семейства `erfc()` возвращает функцию ошибок дополнительную² от аргумента *arg*.

Зависимые функции

`erf()`

Семейство функций `exp`

```
#include <math.h>
float expf(float arg);
double exp(double arg);
long double expl(long double arg);
```

Функции `expf()` и `expl()` добавлены в версии C99.

Каждая функция семейства `exp()` возвращает значение экспоненты от аргумента *arg* (число *e*, возведенное в степень, которая равна значению аргумента *arg*).

¹ Интеграл (вероятности) ошибок:

$$\operatorname{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt.$$

Иногда называется просто *интегралом ошибок* или *интегралом вероятности*. В теории вероятности чаще используется не интеграл вероятности, а *интеграл вероятности Гаусса*, или *функция нормального распределения*

$$\Phi(x) = \frac{1}{2} \left[1 + \operatorname{erf}\left(\frac{x}{\sqrt{2}}\right) \right]. \text{ — Прим. ред.}$$

² Дополнительный интеграл вероятности:

$$\operatorname{erfc}(x) = \frac{2}{\sqrt{\pi}} \int_x^\infty e^{-t^2} dt = 1 - \operatorname{erf}(x). \text{ — Прим. ред.}$$

Пример

Данный фрагмент программы выводит число e , округленное до значения 2.718282.

```
printf("e, возведенное в первую степень, приблизительно равно: %f.",  
exp(1.0));
```

Зависимые функции

`exp2()` и `log()`

Семейство функций `exp2`

```
#include <math.h>  
float exp2f(float arg);  
double exp2(double arg);  
long double exp2l(long double arg);
```

Функции `exp2()`, `exp2f()` и `exp2l()` добавлены в версии C99.

Каждая функция семейства `exp2()` возвращает число 2, возведенное в степень *arg*.

Зависимые функции

`exp()` и `log()`

Семейство функций `expm1`

```
#include <math.h>  
float expm1f(float arg);  
double expm1(double arg);  
long double expm1l(long double arg);
```

Функции `expm1()`, `expm1f()` и `expm1l()` добавлены в версии C99.

Каждая функция семейства `expm1()` возвращает уменьшенное на единицу значение числа e , возведенного в степень *arg* (т.е. возвращаемое значение равно $e^{arg} - 1$).

Зависимые функции

`exp()` и `log()`

Семейство функций `fabs`

```
#include <math.h>  
float fabsf(float num);  
double fabs(double num);  
long double fabsl(long double num);
```

Функции `fabsf()` и `fabsl()` добавлены в версии C99.

Каждая функция семейства `fabs()` возвращает абсолютное значение аргумента *num*.

Пример

Данная программа дважды выводит на экран число 1.0.

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

int main(void)
{
    printf("%1.1f %1.1f", fabs(1.0), fabs(-1.0));

    return 0;
}
```

Зависимые функции

`abs()`

Семейство функций `fdim`

```
#include <math.h>
float fdimf(float a, float b);
double fdim(double a, double b);
long double fdiml(long double a, long double b);
```

Функции `fdim()`, `fdimf()` и `fdiml()` добавлены в версии C99.

Каждая функция семейства `fdim()` возвращает нуль, если значение аргумента *a* меньше значения аргумента *b* или равно ему. В противном случае возвращается результат вычисления разности *a-b*.

Зависимые функции

`remainder()` и `remquo()`

Семейство функций `floor`

```
#include <math.h>
float floorf(float num);
double floor(double num);
long double floorl(long double num);
```

Функции `floorf()` и `floorl()` добавлены в версии C99.

Каждая функция семейства `floor()` возвращает наибольшее целое (представленное в виде значения с плавающей точкой), которое меньше значения аргумента *num* или равно ему. Например, при *num*, равном 1.02, функция `floor()` вернет значение 1.0, а при *num*, равном -1.02, — значение -2.0.

Пример

Данный фрагмент программы выводит на экран число 10.

```
printf("%f", floor(10.9));
```

Зависимые функции

`ceil()` и `fmod()`

■ Семейство функций `fma`

```
#include <math.h>
float fmaf(float a, float b, float c);
double fma(double a, double b, double c);
long double fmal(long double a, long double b, long double c);
```

Функции `fma()`, `fmaf()` и `fmal()` определены в версии C99.

Каждая функция семейства `fma()` возвращает значение выражения $a*b+c$. Округление выполняется только один раз, после завершения всей операции.

Зависимые функции

`round()`, `lround()` и `llround()`

■ Семейство функций `fmax`

```
#include <math.h>
float fmaxf(float a, float b);
double fmax(double a, double b);
long double fmaxl(long double a, long double b);
```

Функции `fmax()`, `fmaxf()` и `fmaxl()` определены в версии C99.

Каждая функция семейства `fmax()` возвращает больший из аргументов a и b .

Зависимые функции

`fmin()`

■ Семейство функций `fmin`

```
#include <math.h>
float fminf(float a, float b);
double fmin(double a, double b);
long double fminl(long double a, long double b);
```

Функции `fmin()`, `fminf()` и `fminl()` определены в версии C99.

Каждая функция семейства `fmin()` возвращает меньший из аргументов a и b .

Зависимые функции

`fmax()`

Семейство функций fmod

```
#include <math.h>
float fmodf(float a, float b);
double fmod(double a, double b);
long double fmodl(long double a, long double b);
```

Функции `fmodf()` и `fmodl()` определены в версии C99.

Каждая функция семейства `fmod()` возвращает остаток от деления аргументов a/b .

Пример

Данная программа выводит на экран число 1.0, являющееся остатком деления 10/3.

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

int main(void)
{
    printf("%1.1f", fmod(10.0, 3.0));

    return 0;
}
```

Зависимые функции

`ceil()`, `floor()` и `fabs()`

Семейство функций frexp

```
#include <math.h>
float frexpf(float num, int *exp);
double frexp(double num, int *exp);
long double frexpl(long double num, int *exp);
```

Функции `frexpf()` и `frexpl()` добавлены в версии C99.

Каждая функция семейства `frexp()` разбивает число *num* на мантиссу *mantissa*, значение которой удовлетворяет неравенствам $0.5 \leq \text{mantissa} < 1$, и целый показатель степени числа 2 (он обозначен через *exp*), притом числа *mantissa* и *exp* выбираются так, чтобы выполнялось равенство $\text{num} = \text{mantissa} * 2^{\text{exp}}$. Значение мантиссы возвращается функцией, а значение показателя¹ присваивается переменной, адресуемой указателем *exp*.

¹ Напомним, что представление числа *num* в виде $\text{num} = \text{mantissa} * b^{\text{exp}}$ (здесь *b* — основание системы счисления) называется *представлением с плавающей точкой (запятой)* или *полулогарифмическим представлением*, и что целая часть логарифма называется характеристикой. Так что $\text{exp} = \chi_2(\text{num}) + 1$, где $\chi_2(\text{num}) = \lfloor \log_2(\text{num}) \rfloor$ — характеристика двоичного логарифма. Число *exp* часто называется *порядком* числа *num* (в нормализованном представлении). Заметим также, что терминам *мантисса* и *характеристика* часто придается и иной смысл. Так, по историческим причинам под *мантиссой* часто подразумевают *дробную часть логарифма*; иногда ее называют также *мантиссой логарифма*. Что же касается *характеристики*, то под ней иногда понимают просто *число, которое представляет порядок в представлении с плавающей запятой*. (В этом смысле в большинстве машин характеристика равна порядку, если он положительный; отличия между ними, как правило, обусловлены тем, что представление порядка, который может быть также и неположительным числом, при реализации операций над числами в полулогарифмическом представлении рассматривают как представление неотрицательного числа.) Так что можно сказать, что характеристика в этом смысле — машинное представление порядка числа. *Порядок* в этом контексте называется также иногда *экспонентой*. (Не путайте с экспонентой-функцией!) — *Прим. ред.*

Пример

Данный фрагмент программы выводит число 0.625 в качестве мантиссы и число 4 — в качестве показателя степени.

```
int e;
double f;

f = frexp(10.0, &e);
printf("%f %d", f, e);
```

Зависимые функции

ldexp()

■ Семейство функций hypot

```
#include <math.h>
float hypotf(float side1, float side2);
double hypot(double side1, double side2);
long double hypotl(long double side1, long double side2);
```

Функции `hypot()`, `hypotf()` и `hypotl()` определены в версии C99.

Каждая функция семейства `hypot()` возвращает длину гипотенузы при заданных длинах двух катетов (т.е. функция возвращает значение квадратного корня из суммы квадратов значений аргументов *side1* и *side2*)¹.

Зависимые функции

sqrt()

■ Семейство функций ilogb

```
#include <math.h>
int ilogbf(float num);
int ilogb(double num);
int ilogbl(long double num);
```

Функции `ilogb()`, `ilogbf()` и `ilogbl()` добавлены в версии C99.

Каждая функция семейства `ilogb()` возвращает порядок аргумента *num*. Возвращаемое значение имеет тип `int`.

Зависимые функции

logb()

■ Семейство функций ldexp

```
#include <math.h>
float ldexpf(float num, int exp);
```

¹ Или расстояние точки с координатами (*side1*; *side2*) от начала координат. — Прим. ред.

```
double ldexp(double num, int exp);
long double ldexpl(long double num, int exp);
```

Функции `ldexpf()` и `ldexpl()` добавлены в версии C99.

Каждая функция семейства `ldexp()` возвращает значение выражения $num * 2^{exp}$.

Пример

Данная программа выводит число 4.

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

int main(void)
{
    printf("%f", ldexp(1, 2));

    return 0;
}
```

Зависимые функции

`frexp()` и `modf()`

Семейство функций lgamma

```
#include <math.h>
float lgammaf(float arg);
double lgamma(double arg);
long double lgammal(long double arg);
```

Функции `lgamma()`, `lgammaf()` и `lgammal()` добавлены в версии C99.

Каждая функция семейства `lgamma()` вычисляет абсолютное значение *гамма-функции*¹ от аргумента *arg* и возвращает ее натуральный логарифм.

Зависимые функции

`tgamma()`

Семейство функций llrint

```
#include <math.h>
long long int llrintf(float arg);
long long int llrint(double arg);
long long int llrintl(long double arg);
```

Функции `llrint()`, `llrintf()` и `llrintl()` добавлены в версии C99.

Каждая функция семейства `llrint()` возвращает значение аргумента *arg*, округленного до ближайшего целого, которое имеет тип `long long int`.

Зависимые функции

`lrint()` и `rint()`

¹ Другие названия: *Г-функция*, *Г-функция Эйлера*, *эйлеров интеграл второго рода*. — Прим. ред.

Семейство функций llround

```
#include <math.h>
long long int llroundf(float arg);
long long int llround(double arg);
long long int llroundl(long double arg);
```

Функции `llround()`, `llroundf()` и `llroundl()` добавлены в версии C99.

Каждая функция семейства `llround()` возвращает значение аргумента *arg*, округленное до ближайшего целого, которое имеет тип `long long int`. Значения, отстоящие от большего и меньшего целых на одинаковую величину (например, число 3.5), округляются в сторону большего целого.

Зависимые функции

`lround()` и `round()`

Семейство функций log

```
#include <math.h>
float logf(float num);
double log(double num);
long double logl(long double num);
```

Функции `logf()` и `logl()` добавлены в версии C99.

Каждая функция семейства `log()` возвращает значение натурального логарифма от аргумента *num*. Если значение аргумента *num* отрицательно, возникает ошибка из-за выхода за пределы области допустимых значений (ошибка из-за нарушения области определения). Если же значение *num* равно нулю, возможна ошибка из-за выхода за пределы диапазона представимых значений.

Пример

Следующая программа выводит на экран значения натуральных логарифмов чисел от 1 до 10 (с шагом 1), т.е. составляет таблицу натуральных логарифмов целых чисел от 1 до 10.

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

int main(void)
{
    double val = 1.0;

    do {
        printf("%f %f\n", val, log(val));
        val++;
    } while (val<11.0);

    return 0;
}
```

Зависимые функции

`log10()` и `log2()`

Семейство функций `log1p`

```
#include <math.h>
float log1pf(float num);
double log1p(double num);
long double log1pl(long double num);
```

Функции `log1p()`, `log1pf()` и `log1pl()` добавлены в версии C99.

Каждая функция семейства `log1p()` возвращает значение натурального логарифма от аргумента *num*+1. Если значение аргумента *num* отрицательно, возникает ошибка из-за выхода за пределы области допустимых значений (ошибка из-за нарушения области определения). Если же значение *num* равно -1, возможна ошибка из-за выхода за пределы диапазона представимых значений.

Зависимые функции

`log()`

Семейство функций `log10`

```
#include <math.h>
float log10f(float num);
double log10(double num);
long double log10l(long double num);
```

Функции `log10f()` и `log10l()` добавлены в версии C99.

Каждая функция семейства `log10()` возвращает значение логарифма по основанию 10 от аргумента *num*. Если значение аргумента *num* отрицательно, возникает ошибка из-за выхода за пределы области допустимых значений (ошибка из-за нарушения области определения). Если же значение *num* равно нулю, возможна ошибка из-за выхода за пределы диапазона представимых значений.

Пример

Данная программа выводит значение десятичных логарифмов чисел, изменяющихся от 1 до 10 с шагом 1, т.е. составляет таблицу десятичных логарифмов целых чисел от 1 до 10.

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

int main(void)
{
    double val = 1.0;

    do {
        printf("%f %f\n", val, log10(val));
        val++;
    } while (val<11.0);

    return 0;
}
```

Зависимые функции

`log()` и `log2()`

Семейство функций log2

```
#include <math.h>
float log2f(float num);
double log2(double num);
long double log2l(long double num);
```

Функции `log2()`, `log2f()` и `log2l()` добавлены в версии C99.

Каждая функция семейства `log2()` возвращает значение логарифма по основанию 2 от аргумента *num*. Если значение аргумента *num* отрицательно, возникает ошибка из-за выхода за пределы области допустимых значений (ошибка из-за нарушения области определения). Если же значение *num* равно нулю, возможна ошибка из-за выхода за пределы диапазона представимых значений¹.

Зависимые функции

`log()` и `log10()`

Семейство функций logb

```
#include <math.h>
float logbf(float num);
double logb(double num);
long double logbl(long double num);
```

Функции `logb()`, `logbf()` и `logbl()` добавлены в версии C99.

Каждая функция семейства `logb()` возвращает показатель аргумента *num*. Возвращаемое значение является числом с плавающей точкой. Если значение аргумента *num* равно нулю, возможна ошибка из-за выхода за пределы диапазона представимых значений.

Зависимые функции

`ilogb()`

Семейство функций lrint

```
#include <math.h>
long int lrintf(float arg);
long int lrint(double arg);
long int lrintl(long double arg);
```

Функции `lrint()`, `lrintf()` и `lrintl()` добавлены в версии C99.

Каждая функция семейства `lrint()` возвращает значение аргумента *arg*, округленное до ближайшего целого, которое имеет тип `long int`.

Зависимые функции

`llrint()` и `rint()`

¹ Как известно, в нуле логарифм не определен, но из-за трудностей представления близких к нулю положительных чисел автор придерживается столь осторожных формулировок. — Прим. ред.

Семейство функций lround

```
#include <math.h>
long int lroundf(float arg);
long int lround(double arg);
long int lroundl(long double arg);
```

Функции `lround()`, `lroundf()` и `lroundl()` добавлены в версии C99.

Каждая функция семейства `lround()` возвращает значение аргумента *arg*, округленное до ближайшего целого, которое имеет тип `long int`. Значения, отстоящие от большего и меньшего целых на одинаковую величину (например, число 3.5), округляются в сторону большего целого.

Зависимые функции

`llround()` и `round()`

Семейство функций modf

```
#include <math.h>
float modff(float num, float *i);
double modf(double num, double *i);
long double modfl(long double num, long double *i);
```

Функции `modff()` и `modfl()` добавлены в версии C99.

Каждая функция семейства `modf()` разбивает аргумент *num* на целую и дробную части. Функция возвращает дробную часть и размещает целую часть в переменной, адресуемой параметром *i*.

Пример

Данный фрагмент программы выводит на экран числа 10 и 0.123.

```
double i;
double f;

f = modf(10.123, &i);
printf("%f %f", i, f);
```

Зависимые функции

`frexp()` и `ldexp()`

Семейство функций nan

```
#include <math.h>
float nanf(const char *content);
double nan(const char *content);
long double nanl(const char *content);
```

Функции `nan()`, `nanf()` и `nanl()` добавлены в версии C99.

Каждая функция семейства `nan()` возвращает значение, которое не является числом и которое содержит строку, адресуемую параметром *content*.

Зависимые функции

`isnan()`

Семейство функций `nearbyint`

```
#include <math.h>
float nearbyintf(float arg);
double nearbyint(double arg);
long double nearbyintl(long double arg);
```

Функции `nearbyint()`, `nearbyintf()` и `nearbyintl()` добавлены в версии C99.

Каждая функция семейства `nearbyint()` возвращает значение аргумента *arg*, округленное до ближайшего целого. Однако возвращаемое число представлено в формате с плавающей точкой.

Зависимые функции

`rint()` и `round()`

Семейство функций `nextafter`

```
#include <math.h>
float nextafterf(float from, float towards);
double nextafter(double from, double towards);
long double nextafterl(long double from, long double towards);
```

Функции `nextafter()`, `nextafterf()` и `nextafterl()` добавлены в версии C99.

Каждая функция семейства `nextafter()` возвращает значение, следующее после аргумента *from*, причем выбор следующего значения осуществляется в направлении, задаваемом аргументом *towards*.

Зависимые функции

`nexttoward()`

Семейство функций `nexttoward`

```
#include <math.h>
float nexttowardf(float from, long double towards);
double nexttoward(double from, long double towards);
long double nexttowardl(long double from, long double towards);
```

Функции `nexttoward()`, `nexttowardf()` и `nexttowardl()` добавлены в версии C99.

Каждая функция семейства `nexttoward()` возвращает значение, следующее после аргумента *from*, причем выбор следующего значения осуществляется в направлении, задаваемом аргументом *towards*. Действие этих функций аналогично действию функций семейства `nextafter()` за исключением того, что параметр всех трех функций *towards* имеет тип `long double`.

Зависимые функции

`nextafter()`

Семейство функций pow

```
#include <math.h>
float powf(float base, float exp);
double pow(double base, double exp);
long double powl(long double base, long double exp);
```

Функции `powf()` и `powl()` добавлены в версии C99.

Каждая функция семейства `pow()` возвращает значение аргумента *base*, возведенное в степень *exp*, т.е. в результате получается $base^{exp}$. Если значение аргумента *base* равно нулю, а *exp* меньше или равно нулю, возможна ошибка из-за выхода за пределы области допустимых значений (ошибка из-за нарушения области определения). Она произойдет также в том случае, если *base* отрицательно, а *exp* не является целым числом. При этом также может возникнуть ошибка из-за выхода за пределы диапазона представимых значений.

Пример

Следующая программа выводит первые десять степеней числа 10, т.е. составляет таблицу степеней числа 10.

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

int main(void)
{
    double x = 10.0, y = 0.0;

    do {
        printf("%f\n", pow(x, y));
        y++;
    } while(y<11.0);

    return 0;
}
```

Зависимые функции

`exp()`, `log()` и `sqrt()`

Семейство функций remainder

```
#include <math.h>
float remainderf(float a, float b);
double remainder(double a, double b);
long double remainderl(long double a, long double b);
```

Функции `remainder()`, `remainderf()` и `remainderl()` определены в версии C99.

Каждая функция семейства `remainder()` возвращает остаток от деления значений аргументов *a/b*.

Зависимые функции

`remquo()`

■ Семейство функций `remquo`

```
#include <math.h>
float remquo(float a, float b, int *quo);
double remquo(double a, double b, int *quo);
long double remquo(long double a, long double b, int *quo);
```

Функции `remquo()`, `remquof()` и `remquol()` определены в версии C99.

Каждая функция семейства `remquo()` возвращает остаток от деления значений аргументов a/b . При этом целое, адресуемое параметром *quo*, будет содержать частное.

Зависимые функции

`remainder()`

■ Семейство функций `rint`

```
#include <math.h>
float rintf(float arg);
double rint(double arg);
long double rintl(long double arg);
```

Функции `rint()`, `rintf()` и `rintl()` добавлены в версии C99.

Каждая функция семейства `rint()` возвращает значение аргумента *arg*, округленное до ближайшего целого. Однако возвращаемое число представлено в формате с плавающей точкой. Может возникнуть исключение вещественного типа.

Зависимые функции

`nearbyint()` и `round()`

■ Семейство функций `round`

```
#include <math.h>
float roundf(float arg);
double round(double arg);
long double roundl(long double arg);
```

Функции `round()`, `roundf()` и `roundl()` добавлены в версии C99.

Каждая функция семейства `round()` возвращает значение аргумента *arg*, округленное до ближайшего целого. Однако возвращаемое число представлено в формате с плавающей точкой. Значения, отстоящие от большего и меньшего целого на одинаковую величину (например, число 3.5), округляются в сторону большего целого.

Зависимые функции

`lround()` и `llround()`

Семейство функций `scalbn`

```
#include <math.h>
float scalbnf(float val, long int exp);
double scalbn(double val, long int exp);
long double scalbnl(long double val, long int exp);
```

Функции `scalbn()`, `scalbnf()` и `scalbnl()` добавлены в версии C99.

Каждая функция семейства `scalbn()` возвращает произведение параметра *val* и значения `FLT_RADIX`, возведенного в степень, которая равна значению параметра *exp*, т.е. в результате получается $val * FLT_RADIX^{exp}$.

Макрос `FLT_RADIX` определен в заголовке `<float.h>`, и его значение равно основанию системы счисления, используемой для представления вещественных чисел.

Зависимые функции

`scalbn()`

Семейство функций `scalbn`

```
#include <math.h>
float scalbnf(float val, int exp);
double scalbn(double val, int exp);
long double scalbnl(long double val, int exp);
```

Функции `scalbn()`, `scalbnf()` и `scalbnl()` добавлены в версии C99.

Каждая функция семейства `scalbn()` возвращает произведение параметра *val* и значения `FLT_RADIX`, возведенного в степень *exp*, т.е. в результате получается $val * FLT_RADIX^{exp}$.

Макрос `FLT_RADIX` определен в заголовке `<float.h>`, и его значение равно основанию системы счисления, используемой для представления вещественных чисел.

Зависимые функции

`scalbn()`

Семейство функций `sin`

```
#include <math.h>
float sinf(float arg);
double sin(double arg);
long double sinl(long double arg);
```

Функции `sinf()` и `sinl()` добавлены в версии C99.

Каждая функция семейства `sin()` возвращает значение синуса от аргумента *arg*. Значение аргумента должно быть задано в радианах.

Пример

Данная программа выводит синусы последовательности значений, лежащих в пределах от -1 до 1 и изменяющихся с шагом одна десятая, т.е. составляет таблицу синусов чисел от -1 до 1.

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

int main(void)
{
    double val = -1.0;

    do {
        printf("Синус %f равен %f.\n", val, sin(val));
        val += 0.1;
    } while(val<=1.0);

    return 0;
}
```

Зависимые функции

asin(), acos(), atan2(), atan(), tan(), cos(), sinh(), cosh() и tanh()

Семейство функций sinh

```
#include <math.h>
float sinhf(float arg);
double sinh(double arg);
long double sinh1(long double arg);
```

Функции `sinhf()` и `sinh1()` добавлены в версии C99.

Каждая функция семейства `sinh()` возвращает значение гиперболического синуса от аргумента *arg*.

Пример

Данная программа выводит гиперболические синусы последовательности значений, лежащих в пределах от -1 до 1 и изменяющихся с шагом одна десятая, т.е. составляет таблицу гиперболических синусов чисел от -1 до 1.

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

int main(void)
{
    double val = -1.0;

    do {
        printf("Гиперболический синус %f равен %f.\n", val, sinh(val));
        val += 0.1;
    } while(val<=1.0);

    return 0;
}
```

Зависимые функции

asin(), acos(), atan2(), atan(), tan(), cos(), cosh() и sin()

Семейство функций sqrt

```
#include <math.h>
float sqrtf(float num);
double sqrt(double num);
long double sqrtl(long double num);
```

Функции `sqrtf()` и `sqrtl()` добавлены в версии C99.

Каждая функция семейства `sqrt()` возвращает значение квадратного корня от аргумента *num*. Если значение аргумента отрицательно, возникает ошибка из-за выхода за пределы области допустимых значений (ошибка из-за нарушения области определения).

Пример

Данный фрагмент программы выводит на экран число 4.

```
printf("%f", sqrt(16.0));
```

Зависимые функции

`exp()`, `log()` и `pow()`

Семейство функций tan

```
#include <math.h>
float tanf(float arg);
double tan(double arg);
long double tanl(long double arg);
```

Функции `tanf()` и `tanl()` добавлены в версии C99.

Каждая функция семейства `tan()` возвращает значение тангенса от аргумента *arg*. Значение аргумента должно быть задано в радианах.

Пример

Данная программа выводит тангенсы последовательности значений, лежащих в пределах от -1 до 1 и изменяющихся с шагом одна десятая, т.е. составляет таблицу тангенсов чисел от -1 до 1.

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

int main(void)
{
    double val = -1.0;

    do {
        printf("Тангенс %f равен %f.\n", val, tan(val));
        val += 0.1;
    } while(val <= 1.0);

    return 0;
}
```

Зависимые функции

`acos()`, `asin()`, `atan()`, `atan2()`, `cos()`, `sin()`, `sinh()`, `cosh()` и `tanh()`

Семейство функций `tanh`

```
#include <math.h>
float tanhf(float arg);
double tanh(double arg);
long double tanhl(long double arg);
```

Функции `tanhf()` и `tanhl()` добавлены в версии C99.

Каждая функция семейства `tanh()` возвращает значение гиперболического тангенса от аргумента *arg*.

Пример

Данная программа выводит гиперболические тангенсы последовательности значений, лежащих в пределах от -1 до 1 и изменяющихся с шагом одна десятая, т.е. составляет таблицу гиперболических тангенсов чисел от -1 до 1.

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

int main(void)
{
    double val = -1.0;

    do {
        printf("Гиперболический тангенс %f равен %f.\n", val, tanh(val));
        val += 0.1;
    } while(val <= 1.0);

    return 0;
}
```

Зависимые функции

`acos()`, `asin()`, `atan()`, `atan2()`, `cos()`, `sin()`, `cosh()`, `sinh()` и `tan()`

Семейство функций `tgamma`

```
#include <math.h>
float tgammaf(float arg);
double tgamma(double arg);
long double tgammal(long double arg);
```

Функции `tgamma()`, `tgammaf()` и `tgammal()` добавлены в версии C99.

Каждая функция семейства `tgamma()` возвращает значение гамма-функции от аргумента *arg*.

Зависимые функции

`lgamma()`

Семейство функций trunc

```
#include <math.h>
float truncf(float arg);
double trunc(double arg);
long double trunc1(long double arg);
```

Функции `trunc()`, `truncf()` и `trunc1()` добавлены в версии C99.

Каждая функция семейства `trunc()` возвращает усеченное значение аргумента *arg*, т.е. значение, в котором отброшена дробная часть¹.

Зависимые функции

`nearbyint()`

¹ Иногда говорят, что это округленное значение аргумента *arg*, причем округление в данном случае выполняется отбрасыванием дробной части. — *Прим. ред.*

Полный
справочник по



Глава 16

**Функции времени, даты
и локализации**

В библиотеке стандартных функций несколько функций предназначено для работы с датой и временем. В ней также определены функции, которые обрабатывают геополитическую информацию, связанную с программой. Приведем описание этих функций.

Для использования функций времени и даты необходим заголовочный файл `<time.h>`. Этот файл определяет три типа данных, связанных с исчислением времени: `clock_t`, `time_t`, и `tm`. Типы данных `clock_t` и `time_t` предназначены для представления системного времени и даты в виде некоторого целого значения, называемого *календарным временем*. Структурный тип `tm` содержит дату и время, разбитые на составляющие компоненты. Структура `tm` состоит из следующих членов:

```
int tm_sec; /* секунды, 0-60 */
int tm_min; /* минуты, 0-59 */
int tm_hour; /* часы, 0-23 */
int tm_mday; /* день месяца, 1-31 */
int tm_mon; /* месяцы, начиная с января, 0-11 */
int tm_year; /* годы, начиная с 1900 */
int tm_wday; /* дни, начиная с воскресенья, 0-6 */
int tm_yday; /* дни, начиная с 1 января, 0-365 */
int tm_isdst /* индикатор летнего времени */
```

Значение `tm_isdst` положительно, если действует режим летнего времени (Daylight Saving Time), равно нулю, если не действует, и отрицательно, если информация об этом недоступна. Такой формат представления времени и даты называется *разделенным на компоненты календарным временем* (*broken-down time*).

Кроме того, в `<time.h>` определен макрос `CLOCKS_PER_SEC`, который содержит число тактов системных часов в секунду.

Функции геополитического окружения описаны в заголовочном файле `<locale.h>`. В нем определена структура `lconv`, которая приведена в описании функции `localeconv()`.

Функция `asctime`

```
#include <time.h>
char *asctime(const struct tm *ptr);
```

Функция `asctime` возвращает указатель на строку, которая содержит информацию, сохраняемую в адресуемой параметром `ptr` структуре и имеющую следующую форму:

День_недели месяц дата часы:минуты:секунды год\n\0

Например:

```
Fri Apr 15 12:05:34 2005
```

`Ptr` указывает на структуру, заполняемую функциями `localtime()` или `gmtime()`.

Буфер, используемый функцией `asctime()` для хранения форматированной строки вывода, является статически распределенным массивом символов. Он перезаписывается при каждом вызове функции. Чтобы сохранить содержание строки, скопируйте ее в какую-нибудь другую область памяти.

Пример

Эта программа отображает местное время, определяемое системой:

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

```
int main(void)
{
    struct tm *ptr;
    time_t lt;

    lt = time(NULL);
    ptr = localtime(&lt);
    printf(asctime(ptr));

    return 0;
}
```

Зависимые функции

localtime(), gmtime(), time() и ctime()

Функция clock

```
#include <time.h>
clock_t clock(void);
```

Функция clock() возвращает значение, которое приблизительно соответствует времени работы вызывающей программы. Для преобразования этого значения в секунды нужно разделить его на значение CLOCKS_PER_SEC. Если системный таймер недоступен, возвращается значение -1.

Пример

Следующая функция отображает время выполнения вызывающей программы в секундах:

```
void elapsed_time(void)
{
    printf("Прошло: %u секунд.\n", clock()/CLOCKS_PER_SEC);
}
```

Зависимые функции

time(), asctime() и ctime()

Функция ctime

```
#include <time.h>
char *ctime(const time_t *time);
```

Функция ctime() возвращает указатель на строку, имеющую следующий вид:

День месяц год часы:минуты:секунды year\n\0.

Функции передается указатель на календарное время. Календарное время обычно получают с помощью функции time().

Буфер, используемый ctime() для хранения форматированной строки вывода является статически распределенным массивом символов. Он перезаписывается при каждом вызове функции. Для сохранения строки скопируйте ее в какую-нибудь другую область памяти.

Пример

Эта программа отображает местное время, определенное в системе:

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

int main(void)
{
    time_t lt;

    lt = time(NULL);

    printf(ctime(&lt));

    return 0;
}
```

Зависимые функции

localtime(), gmtime(), time() и asctime()



Функция difftime

```
#include <time.h>
double difftime(time_t time2, time_t time1);
```

Функция `difftime()` возвращает разность в секундах между значениями параметров `time1` и `time2`, т.е. возвращается значение выражения `time2-time1`.

Пример

Эта программа отображает время в секундах, требуемое для выполнения пустого цикла 5 000 000 раз:

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

int main(void)
{
    time_t start, end;
    volatile long unsigned t;

    start = time(NULL);
    for(t=0; t<5000000; t++);
    end = time(NULL);
    printf("Цикл использовал %f секунд.\n", difftime(end, start));

    return 0;
}
```

Зависимые функции

localtime(), gmtime(), time() и asctime()

■ Функция gmtime

```
#include <time.h>
struct tm *gmtime(const time_t *time);
```

Функция `gmtime()` возвращает указатель на структуру `tm`, содержащую календарное время в разделенной на компоненты форме. Значение `time` представлено в виде так называемого координированного всемирного времени (Coordinated Universal Time, или UTC)¹, которое, по сути, является средним временем по гринвичскому меридиану² (Greenwich mean time, GMT). Функция `time()` возвращает указатель `time`. Она возвращает `NULL`, если система не поддерживает координированное всемирное время.

Память для структуры, в которой функция `gmtime()` сохраняет разделенное на компоненты время, распределяется статически. Эта структура перезаписывается при каждом вызове функции. Чтобы сохранить содержимое структуры, скопируйте ее в какую-нибудь другую область памяти.

Пример

Эта программа печатает местное время и координированное всемирное время (UTC) системы:

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

/* Печать местного и координированного всемирного (UTC) времени. */
int main(void)
{
    struct tm *local, *gm;
    time_t t;

    t = time(NULL);
    local = localtime(&t);
    printf(" Местное время и дата: %s\n", asctime(local));
    gm = gmtime(&t);
    printf("Координированное всемирное время и дата: %s", asctime(gm));

    return 0;
}
```

Зависимые функции

`localtime()`, `time()` и `asctime()`

■ Функция localeconv

```
#include <locale.h>
struct lconv *localeconv(void)
```

¹ Называется также *всеобщим скоординированным временем* и *универсальным глобальным временем (по Гринвичу)*. — *Прим. ред.*

² Называется также *всемирным (гринвичским средним) временем* или *средним временем по Гринвичу*. UTC не может отличаться от GMT более чем на 0,9 с. — *Прим. ред.*

Функция `localeconv()` возвращает указатель на структуру типа `lconv`, которая содержит различную информацию о геополитической среде, связанную со способом форматирования чисел. Структура `lconv` включает следующие члены:

```
char decimal_point;      /* Символ десятичной точки
                           для неденежных значений. */
char thousands_sep;      /* Разделитель тысяч для
                           неденежных значений. */
char grouping;           /* Определяет группирование
                           для неденежных значений. */
char int_curr_symbol;     /* Символ международной валюты. */
char currency_symbol;     /* Символ местной валюты. */
char mon_decimal_point;   /* Символ десятичной точки
                           для денежных значений. */
char mon_thousands_sep; /* Разделитель тысяч для
                           денежных значений. */
char mon_grouping;        /* Определяет группирование
                           для денежных значений. */
char positive_sign;       /* Индикатор положительных
                           денежных значений. */
char negative_sign;       /* Индикатор отрицательных
                           денежных значений. */
char int_frac_digits;     /* Количество цифр
                           справа от десятичной точки для
                           денежных значений, отображаемых
                           в международном формате. */
char frac_digits;         /* Количество цифр справа
                           от десятичной точки для
                           денежных значений, отображаемых
                           в местном формате. */
char p_cs_precedes;       /* 1 - если символ валюты предшествует
                           положительному значению, 0 - если
                           символ валюты следует за значением. */
char p_sep_by_space;      /* 1 - если символ валюты
                           отделяется от значения пробелом,
                           0 - в противном случае. В версии C99
                           содержит разделитель. */
char n_cs_precedes;       /* 1 - если символ валюты предшествует
                           отрицательному значению, 0 - если символ
                           валюты следует за значением. */
char n_sep_by_space;      /* 1 - если символ валюты
                           отделяется от отрицательного
                           значения пробелом, 0 - если
                           символ валюты следует за значением.
                           В версии C99
                           содержит разделитель. */
char p_sign_posn;         /* Указывает позицию символа
                           положительного значения. */
char n_sign_posn;         /* Указывает позицию символа
                           отрицательного значения. */

/* Следующие члены добавлены в C99. */
char _p_cs_precedes;      /* 1 - если символ валюты предшествует
                           положительному значению, 0 - если
                           символ валюты следует за значением.
                           Применяется для значений в международном
                           формате. */
```

```

char _p_sep_by_space; /* Разделитель между символом валюты,
                        знаком и положительным значением.
                        Применяется для значений в международном
                        формате. */
char _n_cs_precedes; /* 1 - если символ валюты предшествует
                        отрицательному значению, 0 - если
                        символ валюты следует за значением.
                        Применяется для значений в международном
                        формате. */
char _n_sep_by_space; /* Разделитель между символом валюты, знаком
                        и отрицательным значением. Применяется для
                        значений в международном формате. */
char _p_sign_posn; /* Указывает позицию символа
                    положительного значения. Применяется
                    для значений в международном формате. */
char _n_sign_posn; /* Указывает позицию символа
                    отрицательного значения. Применяется
                    для значений в международном формате. */

```

Функция `localeconv()` возвращает указатель на структуру `lconv`. Следует помнить, что содержимое этой структуры изменять нельзя. Обратитесь к документации вашего транслятора для определения специфической информации, касающейся структуры `lconv`.

Пример

Следующая программа отображает символ десятичной точки, используемый в текущей локализации:

```

#include <stdio.h>
#include <locale.h>

int main(void)
{
    struct lconv lc;

    lc = *localeconv();

    printf("В качестве разделителя целой и дробной части в десятичных
           числах используется символ %s\n", lc.decimal_point);

    return 0;
}

```

Родственная функция

`setlocale()`

Функция `localtime`

```

#include <time.h>
struct tm *localtime(const time_t *time);

```

Функция `localtime()` возвращает указатель на структуру типа `tm`, содержащую время в разделенной на компоненты форме. Время представлено как местное. Указатель `time` обычно получают с помощью функции `time()`.

Память для структуры, в которой `localtime()` сохраняет разделенное на компоненты время, выделяется статически. Поэтому эта структура перезаписывается при каждом вызове функции. Для сохранения содержания структуры, скопируйте ее в какую-нибудь другую область памяти.

Пример

Эта программа печатает местное время и координированное всемирное время системы:

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

/* Печатать местное и координированное всемирное (UTC) время. */
int main(void)
{
    struct tm * local;
    time_t t;

    t = time(NULL);
    local = localtime(&t);
    printf("Местное время и дата: %s\n", asctime(local));
    local = gmtime(&t);
    printf("Координированное всемирное время (UTC) и дата: %s\n",
          asctime(local));

    return 0;
}
```

Зависимые функции

`gmtime()`, `time()` и `asctime()`



Функция `mktime`

```
#include <time.h>
time_t mktime(struct tm *time);
```

Функция `mktime()` возвращает календарный эквивалент времени, хранящийся в разделенном на компоненты виде в структуре, которая адресуется параметром-указателем `time`. Элементы `tm_wday` и `tm_yday` устанавливаются самой функцией, поэтому их не нужно определять при ее вызове.

Если `mktime()` не может представить информацию в виде допустимого календарного времени, возвращается `-1`.

Пример

Эта программа сообщает день недели 3 января 2005 года:

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

int main(void)
{
    struct tm t;
    time_t t_of_day;
```

```

t.tm_year = 2005-1900;
t.tm_mon = 0;
t.tm_mday = 3;
t.tm_hour = 0; /* Час, минута, секунда не имеют значения, */
t.tm_min = 0; /* если только они не определяют переход */
t.tm_sec = 1; /* на новую дату */
t.tm_isdst = 0;

t_of_day = mktime(&t);
printf(ctime(&t_of_day));

return 0;
}

```

Зависимые функции

time(), gmtime(), asctime() и ctime()

■ Функция setlocale

```

#include <locale.h>
char *setlocale(int type, const char *locale);

```

Функция `setlocale()` позволяет получить или установить некоторые параметры, которые зависят от геополитической среды выполнения программы. Если указатель *locale* является нулем, функция `setlocale()` возвращает указатель на строку текущей локализации. В противном случае функция `setlocale()` попытается использовать строку *locale* для установки локальных параметров в соответствии с параметром *type*. Для задания стандартных C-параметров региональной привязки используйте строку "C", а для задания собственных параметров среды — пустую строку (""). Чтобы получить подробную информацию о строках локализации, поддерживаемых конкретным компилятором, обратитесь к документации.

При вызове функции `setlocale()` в качестве параметра *type* должен быть использован один из следующих макросов (определенных в заголовке `<locale.h>`).

```

LC_ALL
LC_COLLATE
LC_CTYPE
LC_MONETARY
LC_NUMERIC
LC_TIME

```

Макрос `LC_ALL` относится ко всем категориям локализации. Макрос `LC_COLLATE` оказывает влияние на выполнение функции `strcoll()`. Макрос `LC_CTYPE` изменяет характер работы символьных функций. Макрос `LC_MONETARY` определяет денежный формат. Макрос `LC_NUMERIC` изменяет символ десятичной точки для функций форматированного ввода-вывода. Наконец, макрос `LC_TIME` определяет поведение функции `strftime()`.

Функция `setlocale()` возвращает указатель на строку, связанную с параметром *type*.

Пример

Эта программа отображает текущую установку локализации:

```

#include <locale.h>
#include <stdio.h>

```

```
int main(void)
{
    printf(setlocale(LC_ALL, ""));

    return 0;
}
```

Зависимые функции

localeconv(), time(), strcoll() и strftime()

Функция strftime

```
#include <time.h>
size_t strftime(char *str, size_t maxsize, const char *fmt, const
struct tm *time);
```

Функция `strftime()` помещает информацию о времени и дате (вместе с другой информацией) в строку, адресуемую параметром *str*, в соответствии с командами форматирования, которые содержатся в адресуемой параметром *fmt* строке. Эта функция использует разделенное на компоненты время, на которое указывает указатель *time*. В строку *str* будет помещено не более *maxsize* символов.

В версии C99 к параметрам *str*, *fmt* и *time* применен квалификатор `restrict`.

Работа функции `strftime()` напоминает работу функции `sprintf()` в том, что она распознает набор команд форматирования, которые начинаются со знака процента (%), и помещает отформатированный результат в строку. Команды форматирования используются для задания точного способа представления различных данных времени и даты в параметре *str*. Любые другие символы, содержащиеся в строке форматирования, помещаются в строку *str* без изменений. Время и дата отображаются по местному времени. Команды форматирования перечислены в следующей таблице. Обратите внимание на то, что во многих командах прописные и строчные буквы имеют различную интерпретацию.

Функция `strftime()` возвращает количество символов, помещенных в строку, адресуемую параметром *str*, или нуль при возникновении ошибки.

Код	Замещается
%a	Сокращенное название дня недели
%A	Полное название дня недели
%b	Сокращенное название месяца
%B	Полное название месяца
%c	Стандартная строка даты и времени
%C	Две последние цифры года
%d	День месяца в виде десятичного числа (1-31)
%D	Дата в виде месяц/день/год (добавлено в версии C99)
%e	День месяца в виде десятичного числа (1-31) в двух-символьном поле (добавлено в C99)
%F	Дата в виде "год-месяц-день" (добавлено в C99)
%g	Последние две цифры года с использованием понедельного года (добавлено в C99)
%G	Год с использованием понедельного года (добавлено в C99)
%h	Сокращенное название месяца (добавлено в C99)
%H	Час (0-23)
%j	Час (1-12)
%j	День года в виде десятичного числа (1-366)

Код	Замещается
%m	Месяц в виде десятичного числа (1-12)
%M	Минуты в виде десятичного числа (0-59)
%p	Разделитель строк (добавлено в C99)
%p	Местный эквивалент АМ (до полудня) или РМ (после полудня)
%g	12-часовое время (добавлено в C99)
%R	Время в виде чч:мм (добавлено в C99)
%S	Секунды в виде десятичного числа (0-60)
%T	Горизонтальная табуляция (добавлено в C99)
%T	Время в виде чч:мм:сс (добавлено в C99)
%u	День недели; понедельник — первый день недели (0-6) (добавлено в C99)
%U	Неделя года; воскресенье — первый день недели (0-53)
%V	Неделя года с использованием понедельного года (добавлено в C99)
%w	День недели в виде десятичного числа (0-6, воскресенье — 0-й день)
%W	Неделя года; понедельник — первый день недели (0-53)
%x	Стандартная строка даты
%X	Стандартная строка времени
%y	Год в виде десятичного числа без столетия (0-99)
%Y	Год в виде десятичного числа, включающего столетие
%z	Сдвиг относительно координированного всемирного (UTC) времени (добавлено в C99)
%Z	Название часового пояса
%%	Знак процента

Версия C99 позволяет использовать в функции `strftime()` определенные команды форматирования с модификаторами `E` и `O`. Модификатор `E` может модифицировать такие команды, как `c`, `C`, `x`, `X`, `y`, `Y`, `d`, `e` и `n`. Модификатор `O` может модифицировать команды: `I`, `m`, `M`, `S`, `u`, `U`, `V`, `w`, `W` и `y`. Использование этих модификаторов приводит к альтернативному представлению отображаемого времени и/или даты. За подробностями обращайтесь к документации, поставляемой вместе с используемым вами компилятором.

Понедельный год используется командами форматирования `%g`, `%G` и `%V`. При таком представлении первым днем недели является понедельник, а первая неделя года должна включать день с датой “4 января”.

Пример

Предположим, что `lt`time указывает на структуру, которая содержит 10:00:00 АМ. Следующая программа печатает: “Сейчас 10 АМ.”:

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

int main(void)
{
    struct tm *ptr;
    time_t lt;
    char str[80];

    lt = time(NULL);
    ptr = localtime(&lt);

    strftime(str, 100, "Сейчас: %H %p.", ptr);
    printf(str);

    return 0;
}
```

Зависимые функции

`time()`, `localtime()` и `gmtime()`



Функция `time`

```
#include <time.h>
time_t time(time_t *time);
```

Функция `time()` возвращает текущее календарное время системы. Если в системе отсчет времени не производится, возвращается значение -1.

Функцию `time()` можно вызывать либо с нулевым указателем, либо с указателем на переменную типа `time_t`. В последнем случае этой переменной будет присвоено календарное время.

Пример

Эта программа отображает местное время, определенное системой:

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

int main(void)
{
    struct tm *ptr;
    time_t lt;

    lt = time(NULL);
    ptr = localtime(&lt);
    printf(asctime(ptr));

    return 0;
}
```

Зависимые функции

`localtime()`, `gmtime()`, `strftime()` и `ctime()`

Полный
справочник по



Глава 17

**Функции динамического
распределения памяти**

В этой главе описаны функции динамического распределения памяти в языке C. Основные среди них — `malloc()` и `free()`. При каждом вызове `malloc()` выделяется часть остающейся свободной памяти. Каждый вызов `free()` возвращает память системе. Область свободной памяти, в которой распределяется память, называется динамически распределяемой областью памяти или *кучей* (*heap*). Прототипы функций динамического распределения памяти находятся в `<stdlib.h>`.

На заметку

Обзор динамического распределения памяти вы найдете в главе 5.

В стандарте языка C определено четыре функции динамического распределения памяти, которые поддерживаются всеми трансляторами: `calloc()`, `malloc()`, `free()` и `realloc()`. Однако конкретный транслятор почти наверняка содержит несколько версий этих функций, в которых учтены различные возможности и особенности среды. Например, с трансляторами, генерирующими код для сегментированной модели памяти процессора 8086, поставляются специфические функции распределения. Для получения подробных сведений и описания дополнительных функций распределения памяти обратитесь к документации по компилятору.

Функция `calloc`

```
#include <stdlib.h>
void *calloc(size_t num, size_t size);
```

Функция `calloc()` выделяет память, размер которой равен значению выражения `num * size`, т.е. память, достаточную для размещения массива, содержащего `num` объектов размером `size`. Все биты распределенной памяти инициализируются нулями.

Функция `calloc()` возвращает указатель на первый байт выделенной области памяти. Если для удовлетворения запроса нет достаточного объема памяти, возвращается нулевой указатель. Перед попыткой использовать распределенную память важно проверить, что возвращаемое значение не равно нулю.

Пример

Эта функция возвращает указатель на динамически распределенный блок памяти для массива из 100 чисел типа `float`:

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

float *get_mem(void)
{
    float *p;
    p = calloc(100, sizeof(float));
    if(!p) {
        printf("Ошибка при распределении памяти\n");
        exit(1);
    }
    return p;
}
```

Зависимые функции

`free()`, `malloc()` и `realloc()`



Функция free

```
#include <stdlib.h>
void free(void *ptr);
```

Функция `free()` возвращает в динамически распределяемую область памяти блок памяти, адресуемый указателем `ptr`, после чего эта память становится доступной для выделения в будущем.

Обязательно следите, чтобы `free()` вызывалась только с указателем, который был ранее получен в результате вызова одной из системных функций динамического распределения. Использование недопустимого указателя при вызове, скорее всего, приведет к разрушению механизма управления памятью и, возможно, вызовет крах системы. При передаче нулевого указателя функция `free()` не выполняет никакого действия.

Пример

Эта программа распределяет блок памяти для вводимых пользователем строк, а затем освобождает блок памяти:

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

int main(void)
{
    char *str[100];
    int i;

    for(i=0; i<100; i++) {
        if((str[i] = malloc(128))==NULL) {
            printf("Ошибка при распределении памяти\n");
            exit(1);
        }
        gets(str[i]);
    }

    /* Освобождение блока памяти */
    for(i=0; i<100; i++) free(str[i]);

    return 0;
}
```

Зависимые функции

`calloc()`, `malloc()` и `realloc()`



Функция malloc

```
#include <stdlib.h>
void *malloc(size_t size);
```

Функция `malloc()` возвращает указатель на первый байт области памяти размером `size`, которая была выделена из динамически распределяемой области памяти. Если для удовлетворения запроса в динамически распределяемой области памяти нет достаточного объема памяти, возвращается нулевой указатель. Перед попыткой ис-

пользовать выделенную память всегда проверяйте, что возвращаемое значение не является нулевым указателем. Попытка использовать нулевой указатель обычно приводит к полному отказу системы.

Пример

Эта функция выделяет память для структуры типа `addr`:

```
struct addr {
    char name[40];
    char street[40];
    char city[40];
    char state[3];
    char zip [10];
};

struct addr *get_struct(void)
(
    struct addr *p;

    if((p = malloc(sizeof(struct addr)))==NULL) {
        printf("Ошибка при распределении памяти\n");
        exit(1);
    }
    return p;
}
```

Зависимые функции

`free()`, `realloc()` и `calloc()`

Функция `realloc`

```
#include <stdlib.h>
void *realloc(void *ptr, size_t size);
```

Действие функции `realloc()` в версии C99 немного отличается от ее работы в C89, хотя конечный результат одинаков. В C89 функция `realloc()` изменяет размер блока ранее выделенной памяти, адресуемой указателем `ptr` в соответствии с заданным размером `size`. Значение параметра `size` может быть больше или меньше, чем перераспределяемая область. Функция `realloc()` возвращает указатель на блок памяти, поскольку не исключена необходимость перемещения этого блока (например при увеличении размера блока памяти). В этом случае содержимое старого блока (до `size` байтов) копируется в новый блок.

В версии C99 блок памяти, адресуемый параметром `ptr`, освобождается, а вместо него выделяется новый блок. Содержимое нового блока совпадает с содержимым исходного (по крайней мере совпадают первые `size` байтов). Функция возвращает указатель на новый блок. Причем допускается, чтобы новый и старый блоки начинались с одинакового адреса (т.е. указатель, возвращаемый функцией `realloc()`, может совпадать с указателем, переданным в параметре `ptr`).

Если указатель `ptr` нулевой, функция `realloc()` просто выделяет `size` байтов памяти и возвращает указатель на эту память. Если значение параметра `size` равно нулю, память, адресуемая параметром `ptr`, освобождается.

Если в динамически распределяемой области памяти нет достаточного объема свободной памяти для выделения *size* байтов, возвращается нулевой указатель, а исходный блок памяти остается неизменным.

Пример

Эта программа сначала выделяет блок памяти для 17 символов, копирует в них строку "Это - 17 символов", а затем использует `realloc()` для увеличения размера блока до 18 символов, чтобы разместить в конце точку.

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

int main(void)
{
    char *p;

    p = malloc(17);
    if(!p) {
        printf("Ошибка при распределении памяти\n");
        exit(1);
    }

    strcpy(p, "Это - 17 символов");

    p = realloc(p, 18);
    if(!p) {
        printf("Ошибка при распределении памяти\n");
        exit(1);
    }

    strcat(p, ".");

    printf(p);

    free(p);

    return 0;
}
```

Зависимые функции

`free()`, `malloc()` и `calloc()`

Полный
справочник по



Глава 18

Служебные функции

В библиотеке стандартных функций определен целый ряд так называемых служебных функций. Они осуществляют различные преобразования, обрабатывают списки аргументов переменной длины, выполняют сортировку и поиск, а также генерируют случайные числа. Многие из этих функций описаны в заголовочном файле `<stdlib.h>`. В этом заголовке объявлены типы `div_t` и `ldiv_t`. Значения этих типов возвращаются функциями `div()` и `ldiv()` соответственно. В C99 добавлены тип `lldiv_t` и функция `lldiv()`. Здесь также объявлены типы `size_t` и `wchar_t` и определены следующие макрокоманды:

Макрос	Значение
<code>MB_CUR_MAX</code>	Максимальная длина (в байтах) многобайтового символа
<code>NULL</code>	Нулевой указатель
<code>RAND_MAX</code>	Максимальное значение, которое может вернуть функция <code>rand()</code>
<code>EXIT_FAILURE</code>	Значение, возвращаемое вызывающему процессу при неудачном завершении программы
<code>EXIT_SUCCESS</code>	Значение, возвращаемое вызывающему процессу при успешном завершении программы

Если для вызова некоторой функции необходимо использовать заголовок, отличный от `<stdlib.h>`, об этом будет специально указано в описании функции.



Функция abort

```
#include <stdlib.h>
void abort(void);
```

Функция `abort()` вызывает немедленное аварийное завершение программы. Как правило, буфера файлов не дозаписываются. В средах, которые поддерживают эту функцию, она возвращает вызывающему процессу (обычно им является операционная система) значение (определяемое конкретной реализацией), которое сигнализирует об отказе.

Пример

Эта программа заканчивается, если пользователь вводит A:

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

int main(void)
{
    for (;;)
        if(getchar()=='A') abort();

    return 0;
}
```

Зависимые функции

`exit()` и `atexit()`



Функция abs

```
#include <stdlib.h>
int abs(int num);
```

Функция `abs()` возвращает абсолютное значение целочисленного аргумента *num*.

Пример

Эта функция преобразует введенное пользователем число в его абсолютное значение:

```
int get_abs(void)
{
    char num[80];

    gets(num);
    return abs(atoi(num));
}
```

Зависимая функция

`fabs()`



Функция-макрос assert

```
#include <assert.h>
void assert(int exp);
```

Макрос `assert()`, определенный в заголовке `<assert.h>`, записывает информацию об ошибке в поток `stderr`, а затем прекращает выполнение программы, если выражение *exp* равно нулю. В противном случае макрос `assert()` никаких действий не выполняет. Хотя формат выводимого сообщения зависит от конкретной реализации системы программирования, большинство трансляторов используют сообщение, подобное следующему:

```
Assertion failed: <выражение>, file <имя_файла>, line <номер_строки>
```

В версии C99 отображаемое сообщение также включает имя функции, содержащей макрос `assert()`.

Макрос `assert()` обычно используется, чтобы убедиться в правильном выполнении программы, причем *выражение* составляется таким образом, что оно истинно только при отсутствии ошибок.

Нет необходимости удалять из исходного текста программы операторы `assert()` после отладки программы, потому что если определить макрос `NDEBUG`, то макрос `assert()` будет игнорироваться.

Пример

Этот фрагмент кода проверяет, является ли данное, прочитанное из последовательного порта, ASCII-символом (то есть, не используется ли седьмой бит):

```
/* ... */
ch = read_port();
assert(!(ch & 128)); /*проверяет 7-й бит */
```

Зависимая функция

`abort()`.



Функция `atexit`

```
#include <stdlib.h>
int atexit(void (*func)(void));
```

Функция `atexit()` регистрирует функцию, на которую указывает *func*, как функцию, вызываемую при нормальном завершении программы. (Иными словами, при нормальном завершении программы будет вызвана функция, адресуемая параметром *func*.) Функция `atexit()` возвращает нуль, если задаваемая функция успешно зарегистрирована в качестве функции завершения, а в противном случае она возвращает ненулевое значение.

Вообще может быть зарегистрировано до 32 функций завершения, которые будут вызываться в порядке, обратном порядку регистрации (т.е. функция, зарегистрированная последней, выполнится первой).

Пример

Эта программа печатает "Привет здесь" на экране при ее завершении:

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

void done(void);

int main(void)
{
    if(atexit(done)) printf ("Ошибка в atexit().");

    return 0;
}

void done(void)
{
    printf("Привет здесь");
}
```

Зависимые функции

`exit()` и `abort()`



Функция `atof`

```
#include <stdlib.h>
double atof(const char *str);
```

Функция `atof()` преобразует строку, адресуемую параметром *str*, в значение типа `double`. Эта строка должна содержать допустимое число с плавающей точкой. В противном случае возвращаемое значение не определено.

После числа может следовать любой символ, который не может быть частью допустимого числа с плавающей точкой. Имеются в виду пробелы, символы табуляции и пустой строки, знаки препинания (но не точки) и символы, отличные от буквы "Е" или "е". Это значит, что, если функция `atof()` вызывается с аргументом "100.00HELLO", будет возвращено значение 100.00.

Пример

Эта программа читает два числа с плавающей точкой и выводит их сумму:

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

int main(void)
{
    char num1[80], num2[80];

    printf("Введите первое число: ");
    gets(num1);
    printf("Введите второе число: ");
    gets (num2);
    printf("Сумма: %lf.", atof(num1) + atof(num2));

    return 0;
}
```

Зависимые функции

atoi() и atol().

Функция atoi

```
#include <stdlib.h>
int atoi(const char *str);
```

Функция `atoi()` преобразует строку, адресуемую параметром *str*, в значение типа `int`. Эта строка должна содержать допустимое целое число. В противном случае возвращаемое значение не определено.

После числа может следовать любой символ, который не может быть частью целого числа. Имеются в виду пробелы, символы табуляции и пустой строки, знаки препинания и буквы. Это значит, что, если функция `atoi()` вызывается с аргументом "123.23", будет возвращено целое значение 123, а подстрока ".23" будет проигнорирована.

Пример

Следующая программа считывает два целых числа и выводит их сумму:

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

int main(void)
{
    char num1[80], num2[80];

    printf("Введите первое число: ");
    gets (num1);
    printf("Введите второе число: ");
    gets(num2);
    printf("Сумма: %d.", atoi(num1)+atoi(num2));

    return 0;
}
```

Зависимые функции

`atof()` и `atol()`.



Функция `atol`

```
#include <stdlib.h>
long int atol(const char *str);
```

Функция `atol()` преобразует строку, адресуемую параметром *str*, в значение типа `long int`. Эта строка должна содержать допустимое целое число. В противном случае возвращаемое значение не определено.

После числа может следовать любой символ, который не может быть частью целого числа. Имеются в виду пробелы, символы табуляции и пустой строки, знаки препинания и буквы. Это значит, что, если функция `atoi()` вызывается с аргументом "123.23", будет возвращено длинное целое значение 123L, а подстрока ".23" будет проигнорирована.

Пример

Следующая программа считывает два целых числа в виде строк, преобразует их в два длинных целых числа и выводит их сумму:

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

int main(void)
{
    char num1[80], num2[80];

    printf("Введите первое число: ");
    gets(num1);
    printf("Введите второе число: ");
    gets(num2);
    printf("Сумма: %ld.", atol(num1)+atol(num2));

    return 0;
}
```

Зависимые функции

`atof()`, `atoi()` и `atoll()`



Функция `atoll`

```
#include <stdlib.h>
long long int atoll(const char *str);
```

Функция `atoll()` добавлена в версии C99.

Функция `atoll()` преобразует строку, адресуемую параметром *str*, в значение типа `long long int`. В остальном она аналогична функции `atol()`.

Зависимые функции

`atof()`, `atoi()` и `atol()`



Функция `bsearch`

```
#include <stdlib.h>
void *bsearch(const void *key, const void *buf,
              size_t num, size_t size,
              int (*compare)(const void *, const void *));
```

Функция `bsearch()` выполняет двоичный поиск в отсортированном массиве, адресуемом параметром `buf`, и возвращает указатель на первый член, который совпадает с искомым ключом-значением, адресуемым параметром `key`. Количество элементов в массиве задается параметром `num`, а размер (в байтах) каждого элемента — параметром `size`.

Для сравнения каждого элемента массива с ключом-значением используется функция, адресуемая параметром `compare`. Функция `compare` должна иметь следующее определение.

```
int func_name(const void *arg1, const void *arg2);
```

Она должна возвращать значения, описанные в следующей таблице.

Сравнение	Возвращаемое значение
<code>arg1</code> меньше чем <code>arg2</code>	Меньше нуля
<code>arg1</code> равен <code>arg2</code>	Нуль
<code>arg1</code> больше чем <code>arg2</code>	Больше нуля

Массив должен быть отсортирован в порядке возрастания, чтобы по самому младшему адресу содержался наименьший элемент. Если массив не содержит искомого ключа-значения, возвращается нулевой указатель.

Пример

Следующая программа считывает вводимые с клавиатуры символы и определяет, входят ли они в алфавит:

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

char *alpha = "abcdefghijklmnopqrstuvwxyz";

int comp(const void *ch, const void *s);

int main(void)
{
    char ch;
    char *p;

    printf("Введите символ: ");
    ch = getchar();
    ch = tolower(ch);
    p = (char *) bsearch(&ch, alpha, 26, 1, comp);
    if(p) printf(" %c находится в алфавите\n", *p);
    else printf("не входит в алфавит\n");

    return 0;
}

/* Сравнивает два символа. */
int comp(const void *ch, const void *s)
{
    return *(char *)ch - *(char *)s;
}
```

Зависимая функция

qsort()



Функция div

```
#include <stdlib.h>
div_t div(int numerator, int denominator);
```

Функция `div()` возвращает в структуре типа `div_t` частное и остаток, полученные в результате выполнения операции деления числителя *numerator* на знаменатель *denominator*.

Структура типа `div_t` имеет следующие два поля.

```
int quot; /* частное */
int rem;  /* остаток */
```

Пример

Эта программа выводит частное и остаток от деления 10 на 3:

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

int main(void)
{
    div_t n;

    n = div(10, 3);

    printf("Частное и остаток: %d %d.\n", n.quot, n.rem);

    return 0;
}
```

Зависимые функции

ldiv() и lldiv()



Функция exit

```
#include <stdlib.h>
void exit(int exit_code);
```

Функция `exit()` вызывает немедленное нормальное завершение программы. Это значит, что вызываются функции завершения, зарегистрированные функцией `atexit()`, и любые открытые файлы после дозаписи буферов в них закрываются.

В вызывающий процесс (обычно это операционная система) передается значение параметра *exit_code*, если в данной среде предусмотрена поддержка возможных значений. По соглашению, если параметр *exit_code* равен нулю или значению `EXIT_SUCCESS`, предполагается нормальное завершение программы. Ненулевое значение, или значение `EXIT_FAILURE`, используется для индикации ошибки, определенной конкретной реализацией.

Пример

Эта программа обработки списка рассылки позволяет пользователю сделать выбор из меню. Программа завершается, если введена буква Q.

```
int menu(void)
{
    char choice;

    do {
        printf("Ввод имени (E)\n");
        printf("Удаление имени (D)\n");
        printf("Печать (P)\n");
        printf("Выход (Q)\n");
        choice = getchar();
    } while(!strchr("EDPQ", toupper(choice)));

    if(choice=='Q') exit(0);

    return choice;
}
```

Зависимые функции

atexit(), abort() и _Exit()

Функция _Exit

```
#include <stdlib.h>
void _Exit(int exit_code);
```

Функция _Exit() добавлена в версии C99.

Действие функции _Exit() аналогично действию функции exit() за исключением следующих моментов:

- Не вызываются функции завершения, зарегистрированные функцией atexit().
- Не вызываются обработчики сигналов, зарегистрированные функцией signal().
- Не всегда закрываются открытые файлы и, возможно, они не дозаписываются.

Зависимые функции

atexit(), abort() и exit()

Функция getenv

```
#include <stdlib.h>
char *getenv(const char *name);
```

Функция getenv() возвращает указатель на данные о среде, которые хранятся в строке, адресуемой параметром *name* в таблице характеристик среды, определенной конкретной реализацией. Ваша программа не должна изменять значения, хранящиеся в этой таблице.

Среда программы может включать такие данные, как пути и подключенные устройства. Формат данных определяется конкретной реализацией, поэтому для уточнения деталей необходимо обратиться к руководству пользователя, прилагаемому к компилятору.

Если при вызове функции `getenv()` значение аргумента не совпадает ни с одним из данных в описании среды, возвращается нулевой указатель.

Пример

Предположим, что определенный компилятор поддерживает информацию среды относительно устройств, подключенных к системе, тогда следующий фрагмент возвращает указатель на список устройств:

```
char *p;
/* ... */
p = getevn("DEVICES");
```

Зависимая функция

`system()`



Функция labs

```
#include <stdlib.h>
long labs(long num);
```

Функция `labs()` возвращает абсолютное значение аргумента *num*.

Пример

Приведенная ниже функция преобразует введенное с клавиатуры число в его абсолютное значение:

```
long int get_labs()
{
    char num[80];
    gets(num);
    return labs(atol(num));
}
```

Зависимые функции

`abs()` и `llabs()`



Функция llabs

```
#include <stdlib.h>
long long int llabs(long long int num);
```

Функция `llabs()` добавлена в версии C99.

Функция `llabs()` возвращает абсолютное значение аргумента *num*. Она аналогична функции `labs()`, но работает со значениями типа `long long int`.

Зависимые функции

`abs()` и `labs()`



Функция `ldiv`

```
#include <stdlib.h>
ldiv_t ldiv(long int numerator, long int denominator);
```

Функция `ldiv()` возвращает частное и остаток, полученные в результате деления числителя *numerator* на знаменатель *denominator*, в структуре типа `ldiv_t`.

Структура типа `ldiv_t` имеет следующие два поля.

```
long int quot; /* частное */
long int rem;  /* остаток */
```

Пример

Следующая программа выводит частное и остаток от деления 10 на 3:

```
#include <stdlib.h>
#include <stdio.h>
int main(void)
{
    ldiv_t n;

    n = ldiv(10L, 3L);

    printf("Частное и остаток: %ld %ld.\n", n.quot, n.rem);

    return 0;
}
```

Зависимые функции

`div()` и `lldiv()`



Функция `lldiv`

```
#include <stdlib.h>
lldiv_t lldiv(long long int numerator, long long int denominator);
```

Функция `lldiv()` добавлена в версии C99.

Функция `lldiv()` возвращает в структуре типа `lldiv_t` частное и остаток, полученные в результате деления числителя *numerator* на знаменатель *denominator*. Функция `lldiv()` аналогична функции `ldiv()`, но работает со значениями типа `long long int`.

Структура типа `lldiv_t` имеет следующие два поля:

```
long long int quot; /* частное */
long long int rem;  /* остаток */
```

Зависимые функции

`div()` и `ldiv()`



Функция longjmp

```
#include <setjmp.h>
void longjmp(jmp_buf envbuf, int status);
```

Функция `longjmp()` возобновляет выполнение программы с места последнего обращения к функции `setjmp()`. Таким образом, функции `longjmp()` и `setjmp()` предоставляют средство передачи управления между функциями. Обратите внимание на необходимость включения заголовка `<setjmp.h>`.

Функция `longjmp()` восстанавливает состояние стека, сохраненное в буфере `envbuf` с помощью функции `setjmp()`. В результате выполнение программы возобновляется с оператора, следующего за вызовом функции `setjmp()`. Иначе говоря, компьютер “вводится в заблуждение”: “он считает”, будто управление программой не выходило за пределы функции, которая вызвала функцию `setjmp()`. (Выражаясь образно, функция `longjmp()` подобна многомерной машине пространства-времени. Она позволяет путешествовать во времени, не соблюдая какой бы то ни было последовательности событий: с ее помощью можно вернуться в “покинутый мир”, не обращая внимания на то, что предварительно должен был быть произведен выход из вызванных функций. С ее помощью можно “вернуться домой”, минуя промежуточные пункты. Она “искривляет” время и пространство (памяти) так, что с ее помощью можно попасть в покинутую точку программы, не выполняя нормальный процесс возврата из функции.)

Буфер `envbuf` имеет тип `jmp_buf`, который определен в заголовке `<setjmp.h>`. Этот буфер должен быть заполнен в результате обращения к функции `setjmp()` еще до вызова функции `longjmp()`.

Значение параметра `status` становится возвращаемым значением функции `setjmp()`, и оно используется для того, чтобы определить “происхождение длинного перехода”. Единственным недопустимым значением является ноль. Функция `setjmp()` возвращает ноль в том случае, когда она вызывается непосредственно программой, а не косвенно, т.е. путем выполнения функции `longjmp()`.

Функция `longjmp()` используется в основном для возврата из глубоко вложенного набора функций при возникновении ошибок.

Пример

Эта программа печатает 1 2 3:

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

jmp_buf ebuf;
void f2(void);

int main(void)
{
    int i;

    printf("1 ");
    i = setjmp(ebuf);
    if(i == 0) {
        f2();
        printf("Это не будет напечатано.");
    }
    printf("%d", i);

    return 0;
}
```



```

}
void f2(void)
{
    printf("2 ");
    longjmp(ebuf, 3);
}

```

Зависимая функция

setjmp()

Функция mblen

```

#include <stdlib.h>
int mblen(const char *str, size_t size);

```

Функция `mblen()` возвращает длину (в байтах) многобайтового символа, адресуемого параметром *str*. Учету подлежат только первые *size* символов. При ошибке функция возвращает значение -1.

Если указатель *str* нулевой, функция `mblen()` возвращает ненулевое значение в случае, когда многобайтовые символы имеют кодировку, зависящую от территориально-языковых особенностей. В противном случае возвращается нуль.

Пример

Этот оператор отображает размер многобайтового символа, адресуемого указателем *mb*:

```

printf("%d", mblen(mb, 2));

```

Зависимые функции

mbtowc() и wctomb()

Функция mbstowcs

```

#include <stdlib.h>
size_t mbstowcs(wchar_t *out, const char *in, size_t size);

```

Функция `mbstowcs()` преобразует многобайтовую строку, адресуемую параметром *in*, в строку, состоящую из двухбайтовых символов, и помещает результат в массив, адресуемый параметром *out*. В массиве *out* будет сохранено в памяти только *size* байтов.

В версии C99 к параметрам *out* и *in* применен квалификатор `restrict`.

Функция `mbstowcs()` возвращает количество преобразованных многобайтовых символов. При возникновении ошибки функция возвращает значение -1.

Пример

Этот оператор преобразует первые четыре символа в многобайтовой строке, адресуемой указателем *mb*, и помещает результат в *str*.

```

mbstowcs(str, mb, 4);

```

Зависимые функции

wctombs() и mbtowc()



Функция mbtowc

```
#include <stdlib.h>
int mbtowc(wchar_t *out, const char *in, size_t size);
```

Функция `mbtowc()` преобразует многобайтовый символ, который содержится в массиве, адресуемом параметром *in*, в его двухбайтовый эквивалент и помещает результат в объект, адресуемый параметром *out*. Преобразованию подлежат только первые *size* символов.

В версии C99 к параметрам *out* и *in* применен квалификатор `restrict`.

Функция возвращает количество байтов, помещенных в объект *out*. При возникновении ошибки возвращается значение -1. Если указатель *in* нулевой, функция `mbtowc()` возвращает ненулевое значение в случае, когда многобайтовые символы имеют кодировку, зависящую от территориально-языковых особенностей. В противном случае возвращается нуль.

Пример

Этот оператор преобразует многобайтовый символ в *mbstr* в его двухбайтовый эквивалент символа и помещает результат в массив, адресуемый указателем *widenonn*. (Преобразуются только первые 2 байта *mbstr*.)

```
mbtowc(widenonn, mbstr, 2);
```

Зависимые функции

`mblen()` и `wctomb()`



Функция qsort

```
#include <stdlib.h>
void qsort(void *buf, size_t num, size_t size,
int (*compare) (const void *, const void *));
```

Функция `qsort()` сортирует массив, адресуемый параметром-указателем *buf*. (Для сортировки используется алгоритм быстрой сортировки (алгоритм quicksort), разработанный Ч.Э.Р. Хоаром (C.A.R. Hoare). Быстрая сортировка считается наилучшим алгоритмом сортировки общего назначения.) Количество элементов в массиве задается параметром *num*, а размер (в байтах) каждого элемента — параметром *size*.

Для сравнения двух элементов массива используется функция, передаваемая через параметр *compare*. Функция *compare* должна иметь следующее описание.

```
int func_name(const void *arg1, const void *arg2);
```

Она должна возвращать значения, описанные ниже.

Сравнение	Возвращаемое значение
<i>arg1</i> меньше <i>arg2</i>	Меньше нуля
<i>arg1</i> равен <i>arg2</i>	Нуль
<i>arg1</i> больше <i>arg2</i>	Больше нуля

Массив сортируется в порядке возрастания, т.е. по самому младшему адресу будет записан наименьший элемент.

Пример

Следующая программа сортирует список целых чисел и выводит результат:

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

int num[10] = {
    1, 3, 6, 5, 8, 7, 9, 6, 2, 0
};

int comp(const void *, const void *);

int main(void)
{
    int i;

    printf("Исходный массив: ");
    for(i=0; i<10; i++) printf("%d ", num[i]);

    qsort(num, 10, sizeof(int), comp);

    printf("Отсортированный массив: ");
    for(i=0; i<10; i++) printf("%d ", num[i]);

    return 0;
}

/* Сравнение целых */
int comp(const void *i, const void *j)
{
    return *(int *)i - *(int *)j;
}
```

Зависимая функция

bsearch()

Сортировка в убывающем порядке

Функция-параметр *compare* фактически определяет порядок, используемый при сортировке. Задавая с ее помощью различные порядки на сортируемом множестве, можно получить различные упорядочения исходного массива. Например, чтобы отсортировать массив в порядке убывания (т.е. от большего к меньшему), необходимо в этой функции определить обратный (т.е. дуальный или двойственный) порядок. Для этого достаточно определить функцию, лишь знаком отличающуюся от исходной. Это можно сделать, например, так: $compare1(x,y) = compare(y,x)$ или так: $compare1(x,y) = -compare(x,y)$ ¹.



Функция raise

```
#include <signal.h>
int raise(int signal);
```

Функция `raise()` посылает выполняемой программе сигнал, заданный параметром *signal*. При успешном выполнении возвращается ноль, в противном случае — ненулевое значение. Заметьте: функция использует заголовок `<signal.h>`.

¹ Раздел добавлен редактором перевода. — Прим. ред.

Стандартом языка C определены следующие сигналы (не исключено, что конкретный компилятор поддерживает и некоторые дополнительные сигналы).

Макрос	Значение
SIGABRT	Аномальное завершение работы программы
SIGFPE	Ошибка при выполнении действий над вещественными числами
SIGILL	Недопустимая инструкция
SIGINT	Пользователь нажал комбинацию клавиш <Ctrl+C>
SIGSEGV	Неразрешенный доступ к памяти
SIGTERM	Прекратить выполнение программы

Зависимая функция

signal()



Функция rand

```
#include <stdlib.h>
int rand(void);
```

Функция rand() генерирует последовательность псевдослучайных чисел. При каждом обращении к функции возвращается целое в интервале между нулем и значением RAND_MAX, которое в любой реализации должно быть не меньше числа 32 767.

Пример

Следующая программа отображает 10 псевдослучайных чисел:

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

int main(void)
{
    int i;

    for(i=0; i<10; i++)
        printf("%d ", rand());

    return 0;
}
```

Зависимая функция

srand()



Функция setjmp

```
#include <setjmp.h>
int setjmp(jmp_buf envbuf);
```

Макрос setjmp() сохраняет содержимое системного стека в буфере envbuf для использования в будущем с помощью функции longjmp(). Макрос использует заголовок <setjmp.h>.

Макрос-функция `setjmp()` при инициализации возвращает нуль. Однако `longjmp()` передает аргумент функции `setjmp()`, и именно его значение (всегда отличное от нуля), станет значением `setjmp()` после вызова `longjmp()`. Таким образом, если макрос `setjmp()` выполняется после вызова функции `longjmp()`, он возвращает *значение* аргумента, переданного ему функцией `longjmp()`. Дополнительная информация приведена в описании `longjmp`.

Зависимая функция

`longjmp()`

Функция `signal`

```
#include <signal.h>
void (*signal (int signal, void (*func) (int))) (int);
```

Функция `signal()` регистрирует функцию, переданную через параметр-указатель *func*, в качестве обработчика сигнала, указанного параметром *signal*. Это означает, что функция, переданная через указатель-параметр *func*, будет вызвана тогда, когда программа получит сигнал *signal*. Для использования `signal()` требуется включить заголовок `<signal.h>`.

Значением параметра *func* может быть адрес функции обработчика сигнала или один из следующих макросов, определенных в заголовке `<signal.h>`.

Макрос	Значение
<code>SIG_DFL</code>	Использовать стандартную обработку сигнала
<code>SIG_IGN</code>	Игнорировать сигнал данного типа

Если используется адрес функции, то при получении сигнала будет выполнен заданный обработчик. Для получения дополнительных сведений обратитесь к документации, поставляемой с компилятором.

При успешном выполнении функция `signal()` возвращает адрес ранее определенной функции-обработчика данного сигнала. При ошибке возвращается значение `SIG_ERR` (определенное в заголовке `<signal.h>`).

Зависимая функция

`raise()`

Функция `srand`

```
#include <stdlib.h>
void srand(unsigned seed);
```

Функция `srand()` устанавливает исходное число для последовательности, генерируемой функцией `rand()`. (Функция `rand()` возвращает псевдослучайные числа.)

Часто функция `srand()` используется, чтобы при различных запусках программа могла использовать различные последовательности псевдослучайных чисел, — для этого она должна задавать различные исходные числа. Кроме того, с помощью функции `srand()` можно многократно генерировать одну и ту же последовательность псевдослучайных чисел, — для этого нужно задавать в качестве исходного числа одно и то же значение. Иными словами, чтобы многократно генерировать одну и ту же последовательность псевдослучайных чисел, нужно вызывать данную функцию с одним и тем же значением параметра *seed* до начала генерации этой последовательности.

Пример

Следующая программа использует системное время в качестве параметра `srand()`. Это позволяет инициализировать функцию `rand()` случайным числом.

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

/* Засеивает rand() при помощи системного времени
   и отображает первые 10 чисел.
*/
int main(void)
{
    int i, stime;
    long ltime;

    /* получает текущее календарное время */
    ltime = time(NULL);
    stime = (unsigned) ltime/2;
    srand(stime);

    for(i=0; i<10; i++) printf("%d ", rand());

    return 0;
}
```

Зависимая функция

`rand()`



Функция strtod

```
#include <stdlib.h>
double strtod(const char *start, char **end);
```

Функция `strtod()` преобразует строковое представление числа, которое содержится в строке, адресуемой параметром `start`, в значение типа `double` и возвращает полученный результат.

В версии C99 к параметрам `start` и `end` применен квалификатор `restrict`.

Функция `strtod()` работает следующим образом:

Сначала в строке, адресуемой параметром `start`, пропускаются пробелы, символы табуляции и пустой строки. Затем считываются символы, составляющие число. Когда считывается символ, который не может встречаться в записи числа с плавающей точкой, считывание прекращается. К таким символам относятся пробелы, символы табуляции и пустой строки, знаки препинания (но не точки) и символы, отличные от букв "Е" и "е". Наконец, параметр-указатель `end` устанавливается так, чтобы указывать на "неиспользованный" остаток исходной строки, если таковой существует. Это означает, что, если функция `strtod()` вызывается с аргументом "100.00 плоскогубцев", то она возвратит значение 100.00, а параметр-указатель `end` будет указывать на пробел, предшествующий слову "плоскогубцев".

При возникновении переполнения функция `strtod()` возвращает либо значение `HUGE_VAL`, либо значение `-HUGE_VAL` (означающее положительное или отрицательное переполнение соответственно), а глобальная переменная `errno` устанавливается равной зна-

чению ERANGE, свидетельствующему об ошибке из-за выхода результата за пределы представимых чисел. При потере значимости возвращается нуль, а глобальная переменная `errno` устанавливается равной значению ERANGE. Если параметр *start* не указывает на число, никакого преобразования не выполняется и функция возвращает нуль.

Пример

Следующая программа читает числа с плавающей точкой из массива символов.

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

int main(void)
{
    char *end, *start = "100.00 плоскогубцев 200.00 молотков";

    end = start;
    while(*start) {
        printf("%f, ", strtod(start, &end));
        printf("Остаток: %s\n", end);
        start = end;
        /* пропускает символы, не входящие в числа */
        while(!isdigit(*start) && *start) start++;
    }

    return 0;
}
```

Вот что выводит эта программа:

```
100.000000, Остаток: плоскогубцев 200.00 молотков
200.000000, Остаток: молотков
```

Зависимые функции

`atof()`, `strtold()` и `strtof()`

См. также функции `atof()`, `strtold()` и `strtof()`.

Функция `strtof`

```
#include <stdlib.h>
float strtof(const char * restrict start,
             char restrict ** restrict end);
```

Функция `strtof()` добавлена в версии C99.

Функция `strtof()` аналогична функции `strtod()` за исключением того, что она возвращает значение типа `float`. При возникновении переполнения возвращается либо значение `HUGE_VAL`, либо значение `-HUGE_VAL`, а глобальная переменная `errno` устанавливается равной значению ERANGE, свидетельствующему об ошибке из-за выхода результата за пределы представимых чисел. Если параметр *start* не указывает на число, никакого преобразования не выполняется и функция возвращает нуль.

Зависимые функции

`atof()`, `strtold()` и `strtol()`



Функция strtol

```
#include <stdlib.h>
long int strtol(const char *start, char **end,
int radix);
```

Функция `strtol()` преобразует строковое представление числа, которое содержится в строке, адресуемой параметром-указателем *start*, в значение типа `long int` и возвращает полученный результат. Основание системы счисления, в которой представлено преобразуемое число, определяется параметром *radix*. Если значение *radix* равно нулю, то основание определяется так же, как и основание системы счисления при записи констант. Если значение *radix* не равно нулю, то оно должно быть целым числом от 2 до 36.

В версии C99 к параметрам *start* и *end* применен квалификатор `restrict`.

Функция `strtol()` работает следующим образом:

Сначала в строке, адресуемой параметром *start*, пропускаются пробелы, символы табуляции и пустой строки. Затем считывается число. Считывание заканчивается как только будет обнаружен символ, который не может быть частью длинного целого числа. К таким символам относятся пробелы, символы табуляции и пустой строки, знаки препинания и другие символы. Наконец, параметр *end* устанавливается так, чтобы указывать на “неиспользованный” остаток исходной строки, если таковой существует. Это означает, что, если функция `strtol()` вызывается с аргументом “100 клещей”, она возвратит значение 100L, а параметр *end* будет указывать на пробел, предшествующий слову “клещей”.

Если результат не может быть представлен как значение типа `long int`, функция `strtol()` возвращает либо значение `LONG_MAX`, либо значение `LONG_MIN`, а глобальная переменная `errno` устанавливается равной значению `ERANGE`, свидетельствующему об ошибке из-за выхода за границы представимых чисел. Если параметр *start* не указывает на число, никакого преобразования не выполняется и функция возвращает нуль.

Пример

Следующая функция может использоваться для чтения из стандартного входного потока числа, представленного в десятичной системе счисления. Данная функция возвращает результат (целое число) типа `long`.

```
long int read_long(void)
{
    char start[80], *end;

    printf("Введите число: ");
    gets(start);
    return strtol(start, &end, 10);
}
```

Зависимые функции

`atol()` и `strtoll()`

Функция strtold

```
#include <stdlib.h>
long double strtold(const char * restrict start,
                   char ** restrict end);
```

Функция `strtold()` добавлена в версии C99.

Функция `strtold()` аналогична функции `strtod()` за исключением того, что она возвращает значение типа `long double`. При возникновении переполнения возвращается либо значение `HUGE_VALL`, либо значение `-HUGE_VALL`, а глобальная переменная `errno` устанавливается равной значению `ERANGE`, свидетельствующему об ошибке из-за выхода результата за пределы представимых чисел. Если параметр *start* не указывает на число, никакого преобразования не выполняется и функция возвращает нуль.

Зависимые функции

`atof()`, `strtold()` и `strtod()`

Функция strtoll

```
#include <stdlib.h>
long long int strtoll(const char * restrict start,
                    char ** restrict end, int radix);
```

Функция `strtoll()` добавлена в версии C99.

Функция `strtoll()` аналогична функции `strtol()` за исключением того, что она возвращает значение типа `long long int`. Если результат не может быть представлен как значение типа `long long int`, возвращается либо значение `LLONG_MAX`, либо значение `LLONG_MIN`, а глобальная переменная `errno` устанавливается равной значению `ERANGE`, свидетельствующему об ошибке из-за выхода результата за пределы представимых чисел. Если параметр *start* не указывает на число, никакого преобразования не выполняется и функция возвращает нуль.

Зависимые функции

`atol()` и `strtol()`

Функция strtoul

```
#include <stdlib.h>
unsigned long int strtoul(const char *start, char **end,
                        int radix);
```

Функция `strtoul()` преобразует строковое представление числа, которое содержится в строке, адресуемой параметром *start*, в значение типа `unsigned long` и возвращает полученный результат. Основание системы счисления, в которой представлено число, определяется параметром *radix*. Если значение *radix* равно нулю, то основание определяется так же, как и основание системы счисления при записи констант. Если значение *radix* не равно нулю, то оно должно быть целым числом от 2 до 36.

В версии C99 к параметрам *start* и *end* применен квалификатор `restrict`.

Функция `strtoul()` работает следующим образом:

Сначала в строке, адресуемой параметром *start*, пропускаются пробелы, символы табуляции и пустой строки. Затем считывается число. Считывание заканчивается как только будет обнаружен символ, который не может быть частью длинного целого числа без знака. К таким символам относятся пробелы, символы табуляции и пустой строки, знаки препинания и другие символы. Наконец, параметр *end* устанавливается так, чтобы указывать на "неиспользованный" остаток исходной строки, если такой существует. Например, если функция `strtoul()` вызывается с аргументом "100 клешей", то она возвратит значение 100L, а параметр *end* будет указывать на пробел, предшествующий слову "клешей".

Если результат не может быть представлен как длинное целое без знака, функция `strtoul()` возвращает значение `ULONG_MAX`, а глобальная переменная `errno` устанавливается равной значению `ERANGE`, что свидетельствует об ошибке из-за выхода результата за пределы представимых чисел. Если параметр *start* не указывает на число, никакого преобразования не выполняется и функция возвращает ноль.

Пример

Следующая функция может использоваться для чтения из стандартного входного потока числа, представленного в шестнадцатеричной системе счисления. Данная функция возвращает результат (целое число) типа `unsigned long`.

```
unsigned long int read_unsigned_long(void)
{
    char start[80], *end;

    printf("Введите шестнадцатеричное число: ");
    gets(start);
    return strtoul(start, &end, 16);
}
```

Зависимые функции

`strtol()` и `strtoull()`



Функция `strtoull`

```
#include <stdlib.h>
unsigned long long strtoull(const char *restrict start,
                           char **restrict end, int radix);
```

Функция `strtoull()` добавлена в версии C99.

Функция `strtoull()` аналогична функции `strtoul()` за исключением того, что она возвращает значение типа `unsigned long long int`. Если результат не может быть представлен как значение типа `unsigned long long int`, возвращается значение `ULLONG_MAX`, а глобальная переменная `errno` устанавливается равной значению `ERANGE`, свидетельствующему об ошибке из-за выхода результата за пределы представимых чисел. Если параметр *start* не указывает на число, никакого преобразования не выполняется и функция возвращает ноль.

Зависимые функции

`strtol()` и `strtoul()`



Функция system

```
#include <stdlib.h>
int system(const char *str);
```

Функция `system()` передает строку, адресуемую параметром *str*, в качестве команды для командного процессора операционной системы.

Если функция `system()` вызывается с нулевым указателем, она возвращает ненулевое значение при условии доступности командного процессора и нуль в противном случае. (Программы, выполняемые в специальных средах, могут не иметь доступа к командному процессору.) Значение, возвращаемое функцией `system()`, определяется конкретной реализацией. Но обычно возвращается нуль при успешном выполнении команды, а ненулевое значение кода возврата означает наличие ошибки.

Пример

В операционной системе Windows эта программа отображает содержимое текущего каталога:

```
#include <stdlib.h>

int main(void)
{
    return system("dir");
}
```

Зависимая функция

`exit()`



Функции-макросы `va_arg`, `va_start`, `va_end` и `va_copy`

```
#include <stdarg.h>
type va_arg(va_list argptr, type);
void va_copy(va_list target, va_list source);
void va_end(va_list argptr);
void va_start(va_list argptr, last_parm);
```

Макрос `va_copy()` добавлен в версии C99.

Для передачи функции переменного числа аргументов совместно используются макросы `va_arg`, `va_start` и `va_end`. Самым распространенным примером функции, которая принимает переменное число аргументов, является функция `printf()`. Тип `va_list` определен в заголовке `<stdarg.h>`.

Общая процедура создания функции, которая может принимать переменное количество аргументов, такова:

Функция должна иметь по крайней мере один известный параметр (может и больше), указываемый до переменного списка параметров. (Такие параметры называются также обязательными, а параметры, следующие за ними — необязательными.) Крайний правый известный параметр называется *last_parm*. (Он предшествует первому необязательному параметру.) Его имя используется в качестве второго параметра в обращении к макросу `va_start()`. Чтобы получить доступ к любому дополнительному параметру, сначала не-

обходимо инициализировать указатель-аргумент *argptr*¹, обратившись к макросу *va_start()*. (Иными словами, необходимо выполнить вызов *va_start(argptr, <имя last_parm>)*. — Прим. ред.) После этого значения параметров возвращаются в результате вызова макроса *va_arg()*. В качестве второго аргумента этого макроса (соответствующего параметру *type*), нужно указать тип следующего параметра². Наконец, после прочтения всех параметров до возвращения из функции необходимо вызвать макрос *va_end()*, чтобы гарантировать корректное восстановление стека. Если макрос *va_end()* вызван не будет, высока вероятность аварийного отказа программы.

Макрос *va_copy()* копирует список аргументов, обозначенный параметром *target*, в объект, обозначенный параметром *source*.

Пример

Эта программа использует функцию *sum_series()*, возвращающую сумму последовательности чисел. Первый аргумент содержит число дополнительно передаваемых аргументов. В этом примере программа суммирует первые пять слагаемых суммы

$$\frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \frac{1}{16} + \dots + \frac{1}{2^N}$$

Будет выведено 0.968750.

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

double sum_series(int num. ...);

/* Пример переменного числа аргументов - сумма последовательности. */
int main(void)
{
    double d;

    d= sum_series(5, 0.5, 0.25, 0.125, 0.0625, 0.03125);

    printf("Сумма последовательности %f.\n", d);

    return 0;
}

double sum_series(int num, ...)
{
    double sum=0.0, t;
    va_list argptr;

    /* Инициализация argptr */
    va_start(argptr, num);

    /* сумма последовательности */
    for( ; num; num--) {
        t = va_arg(argptr, double); /* получить следующий аргумент */
        sum += t;
    }
}
```

¹ Он должен быть объявлен, например, так: *va_list argptr*; — Прим. ред.

² Ну где вы видели функцию, которой в качестве аргумента передается служебное слово? Конечно, это возможно именно потому, что *va_arg()* — макрос, а не функция языка C. Кроме того, посмотрите на описание *type va_arg(va_list argptr, type)*; Какой же компилятор позволит вам так описывать функцию?! — Прим. ред.

```

/* выполнение корректного выхода */
va_end(argptr);
return sum;
}

```

Зависимая функция

`vprintf()`

■ Функция `wcstombs`

```

#include <stdlib.h>
size_t wcstombs(char *out, const wchar_t *in, size_t size);

```

Функция `wcstombs()` преобразует массив двухбайтовых символов, адресуемый параметром-указателем *in*, в его многобайтовый эквивалент и помещает результат в массив, адресуемый параметром *out*. Преобразованию подлежат только первые *size* символов. Процесс преобразования прекращается раньше, если будет обнаружен символ конца строки ('0').

В версии C99 к параметрам *out* и *in* применен квалификатор `restrict`.

При успешном выполнении функция `wcstombs()` возвращает количество байтов, помещенных в массив *out*. При возникновении ошибки возвращается значение -1.

Зависимые функции

`wctomb()` и `mbstowcs()`

■ Функция `wctomb`

```

#include <stdlib.h>
int wctomb(char *out, wchar_t in);

```

Функция `wctomb()` преобразует двухбайтовый символ, содержащийся в параметре *in*, в его многобайтовый эквивалент и помещает результат в массив, адресуемый параметром *out*. Массив, адресуемый параметром *out*, должен иметь длину не меньше `MB_CUR_MAX` символов.

При успешном выполнении функция `wctomb()` возвращает количество байтов, содержащихся в многобайтовом символе. При возникновении ошибки возвращается значение -1.

Если параметр *out* равен нулю, функция `wctomb()` возвращает ненулевое значение в случае, когда многобайтовый символ имеет кодировку, зависящую от территориально-языковых особенностей. В противном случае возвращается нуль.

Зависимые функции

`wctomb(s)` и `mbstowcs()`

Полный
справочник по



Глава 19

**Функции обработки
двухбайтовых символов**

В 1995 году к стандарту C89 был добавлен ряд функций, предназначенных для обработки двухбайтовых символов (wide-character functions), которые позже вошли в стандарт C99. Эти функции работают с символами типа `wchar_t` длиной 16 бит. Для большинства этих функций существуют эквивалентные им функции, работающие с символами типа `char`. Например, функция `iswspace()` для обработки двухбайтовых символов является версией функции `isspace()`. В целом, имена функций для обработки двухбайтовых символов образованы из имен аналогичных функций для работы с символами типа `char` путем добавления буквы “w” (wide-character, т.е. “широкоформатный” символ).

В языке C функции, предназначенные для работы с двухбайтовыми символами, используют два заголовка: `<wchar.h>` и `<wctype.h>`. В заголовке `<wctype.h>` определены типы `wint_t`, `wctrans_t` и `wctype_t`. Многие функции, предназначенные для работы с двухбайтовыми символами, принимают в качестве параметра двухбайтовый символ. Такой параметр имеет тип `wint_t`, в которой можно записать двухбайтовый символ. Использование типа `wint_t` в функциях, предназначенных для работы с двухбайтовыми символами, аналогично использованию типа `int` в функциях, обрабатывающих тип `char`. Типы `wctrans_t` и `wctype_t` — это типы объектов, используемые для преобразования символов и определения категории символа соответственно. Кроме того, в заголовке `<wctype.h>` определен двухбайтовый признак конца файла (EOF) под именем `WEOF`.

Помимо `wint_t`, в заголовке `<wchar.h>` определены такие типы, как `wchar_t`, `size_t` и `mbstate_t`. Тип `wchar_t` создает двухбайтовый символ-объект, а `size_t` — это тип значения, возвращаемого оператором `sizeof`. Тип `mbstate_t` описывает объект, который хранит состояние преобразования многобайтового объекта в двухбайтовые символы. Заголовок `<wchar.h>` также определяет макросы `NULL`, `WEOF`, `WCHAR_MAX` и `WCHAR_MIN`. Последние два макроса определяют максимальное и минимальное значения, которые могут храниться в объекте типа `wchar_t`.

Поскольку большинство функций, предназначенных для работы с двухбайтовыми символами, аналогичны соответствующим им функциям для работы с символами типа `char`, т.е. тем самым функциям, которые часто используется большинством программистов, знакомых с языком C, то в этой главе приводится лишь краткое описание таких функций.



Функции классификации двухбайтовых символов

Заголовок `<wctype.h>` содержит прототипы тех функций, которые позволяют классифицировать двухбайтовые символы. Эти функции распределяют по категориям двухбайтовые символы или преобразуют регистр буквенного символа, устанавливая строчное или прописное написание. В табл. 19.1 приведены списки этих функций, а также соответствующие им функции для работы с символами типа `char`, которые были описаны в главе 14.

Таблица 19.1. Функции, предназначенные для работы с двухбайтовыми символами, и соответствующие им функции для типа `char`

Функция	Соответствующая функция для типа <code>char</code>
<code>int iswalnum(wint_t ch)</code>	<code>isalnum()</code>
<code>int iswalpna(wint_t ch)</code>	<code>isalpha()</code>
<code>int iswblank(wint_t ch)</code>	<code>isblank()</code> (Добавлена в C99.)
<code>int iswcntrl(wint_t ch)</code>	<code>iscntrl()</code>
<code>int iswdigit(wint_t ch)</code>	<code>isdigit()</code>

Функция	Соответствующая функция для типа <i>char</i>
<code>int iswgraph(wint_t ch)</code>	<code>isgraph()</code>
<code>int iswlower(wint_t ch)</code>	<code>islower()</code>
<code>int iswprint(wint_t ch)</code>	<code>isprint()</code>
<code>int iswpunct(wint_t ch)</code>	<code>ispunct()</code>
<code>int iswspace(wint_t ch)</code>	<code>isspace()</code>
<code>int iswupper(wint_t ch)</code>	<code>isupper()</code>
<code>int iswxdigit(wint_t ch)</code>	<code>isxdigit()</code>
<code>wint_t towlower(wint_t ch)</code>	<code>tolower()</code>
<code>wint_t toupper(wint_t ch)</code>	<code>toupper()</code>

Помимо функций, приведенных в табл. 19.1, в заголовке `<wctype.h>` определены следующие функции, которые предоставляют открытые средства классификации символов.

```
wctype_t wctype(const char *attr);
int iswctype(wint_t ch, wctype_t attr_ob);
```

Функция `wctype()` возвращает значение, которое можно передать функции `iswctype()` в качестве параметра *attr_ob*. Строка, адресуемая параметром *attr*, задает свойство, которое должен иметь символ. Это значение можно затем использовать для определения, является ли *ch* символом, который обладает этим свойством. Если является, то функция `iswctype()` возвращает ненулевое значение. В противном случае возвращается нуль. В любых условиях выполнения программы определены следующие строки свойств:

<code>alnum</code>	<code>digit</code>	<code>print</code>	<code>upper</code>
<code>alpha</code>	<code>graph</code>	<code>punct</code>	<code>xdigit</code>
<code>cntrl</code>	<code>lower</code>	<code>space</code>	

В версии C99 также определена строка `blank`.

Следующий фрагмент демонстрирует использование функций `wctype()` и `iswctype()`:

```
wctype_t x;

x = wctype("space");

if(iswctype(L' ', x))
    printf("Это пробел.\n");
```

Будет выведено "Это пробел".

Кроме того, в заголовке `<wctype.h>` определены функции `wctrans()` и `towctrans()`. Их описания приведены ниже.

```
wctrans_t wctrans(const char *mapping);
wint_t towctrans(wint_t ch, wctrans_t mapping_ob);
```

Функция `wctrans()` возвращает значение, которое можно передать функции `towctrans()` в качестве параметра *mapping_ob*. Строка, адресуемая параметром *mapping*, определяет отображение одного символа на другой. Данная строка затем может быть использована функцией `towctrans()` для преобразования символа *ch*. Функция возвращает преобразованное значение. При всех условиях выполнения программы поддерживаются следующие строки преобразования.

<code>tolower</code>	<code>toupper</code>
----------------------	----------------------

Следующая последовательность демонстрирует применение функций `wctrans()` и `towctrans()`:

```
wctrans_t x;

x = wctrans("tolower");

wchar_t ch = towctrans(L'W', x);
printf("%c", (char) ch);
```

Выводит `w` на нижнем регистре.



Функции ввода-вывода двухбайтовых символов

Некоторые функции ввода-вывода, описанные в главе 13, имеют реализации, ориентированные на работу с двухбайтовыми символами. Эти функции (они перечислены в табл. 19.2) используют заголовок `<wchar.h>`. Обратите внимание на то, что функции `swprintf()` и `vswprintf()` требуют передачи дополнительного параметра, в котором не нуждаются соответствующие им функции для типа `char`.

Таблица 19.2. Функции ввода/вывода для двухбайтовых символов и соответствующие им функции для типа `char`

Функция	Соответствующая функция для типа <code>char</code>
<code>wint_t fgetwc(FILE *stream)</code>	<code>fgetc()</code>
<code>wchar_t *fgetws(wchar_t *str, int num, FILE *stream)</code>	<code>fgets()</code> В версии C99 к параметрам <i>str</i> и <i>stream</i> применен квалификатор <code>restrict</code>
<code>wint_t fputwc(wchar_t ch, FILE *stream)</code>	<code>fputc()</code>
<code>int fputws(const wchar_t *str, FILE *stream)</code>	<code>fputs()</code> В версии C99 к параметрам <i>str</i> и <i>stream</i> применен квалификатор <code>restrict</code>
<code>int fwprintf(FILE *stream, const wchar_t *fmt, ...)</code>	<code>fprintf()</code> В версии C99 к параметрам <i>stream</i> и <i>fmt</i> применен квалификатор <code>restrict</code>
<code>int fwscanf(FILE *stream, const wchar_t *fmt, ...)</code>	<code>fscanf()</code> В версии C99 к параметрам <i>stream</i> и <i>fmt</i> применен квалификатор <code>restrict</code>
<code>wint_t getwc(FILE *stream)</code>	<code>getc()</code>
<code>wint_t getwchar(void)</code>	<code>getchar()</code>
<code>wint_t putwc(wchar_t ch, FILE *stream)</code>	<code>putc()</code>
<code>wint_t putwchar(wchar_t ch)</code>	<code>putchar()</code>
<code>int swprintf(wchar_t *str, size_t num, const wchar_t *fmt, ...)</code>	<code>sprintf()</code> Обратите внимание на то, что добавлен параметр <i>num</i> , который ограничивает количество символов, записываемых в массив <i>str</i> . В версии C99 к параметрам <i>str</i> и <i>fmt</i> применен квалификатор <code>restrict</code>
<code>int swscanf(const wchar_t *str, const wchar_t *fmt, ...)</code>	<code>sscanf()</code> В версии C99 к параметрам <i>str</i> и <i>fmt</i> применен квалификатор <code>restrict</code>
<code>wint_t ungetwc(wint_t ch, FILE *stream)</code>	<code>ungetc()</code>

Функция	Соответствующая функция для типа <i>char</i>
<code>int vfprintf(FILE *stream, const wchar_t *fmt, va_list arg)</code>	<code>vfprintf()</code> В версии C99 к параметру <i>fmt</i> применен квалификатор <i>restrict</i>
<code>int vfwscanf(FILE * restrict stream, const wchar_t * restrict fmt, va_list arg);</code>	<code>vwscanf()</code> (Добавлена в версии C99.)
<code>int vswprintf(wchar_t *str, size_t num, const wchar_t *fmt, va_list arg)</code>	<code>vswprintf()</code> Обратите внимание на то, что добавлен параметр <i>num</i> , который ограничивает количество символов, записываемых в массив <i>str</i> . В версии C99 к параметрам <i>str</i> и <i>fmt</i> применен квалификатор <i>restrict</i>
<code>int vswscanf(const wchar_t * restrict str, const wchar_t * restrict fmt, va_list arg);</code>	<code>vsscanf()</code> (Добавлена в версии C99.)
<code>int vwprintf(const wchar_t *fmt, va_list arg)</code>	<code>vwprintf()</code> В версии C99 к параметру <i>fmt</i> применен квалификатор <i>restrict</i>
<code>int vwscanf(const wchar_t * restrict fmt, va_list arg);</code>	<code>vwscanf()</code> (Добавлена в версии C99.)
<code>int wprintf(const wchar_t *fmt, ...)</code>	<code>wprintf()</code> В версии C99 к параметру <i>fmt</i> применен квалификатор <i>restrict</i>
<code>int wscanf(const wchar_t *fmt, ...)</code>	<code>wscanf()</code> В версии C99 к параметру <i>fmt</i> применен квалификатор <i>restrict</i>

Дополнительно к функциям, показанным в таблице, добавлена следующая функция, ориентированная на работу с двухбайтовыми символами:

```
int fwide(FILE *stream, int how);
```

Если значение параметра *how* положительно, функция `fwide()` делает поток *stream* потоком двухбайтовых символов. Если же значение параметра *how* отрицательно, то функция `fwide()` превращает поток *stream* в поток объектов типа `char`. А если значение *how* равно нулю, на поток *stream* никакого воздействия не оказывается. Если этот поток уже был ориентирован либо на двухбайтовые, либо на обычные символы, он изменяться не будет. Функция возвращает положительное значение, если поток рассматривается как содержащий двухбайтовые символы. Отрицательное значение возвращается, если он рассматривается как содержащий символы типа `char`. В случае, когда поток еще не ориентирован, функция возвращает нуль. Ориентация потока также определяется его первым использованием.



Функции для операций над строками двухбайтовых символов

Для операций над строками двухбайтовых символов существуют версии функций, описанных в главе 14. Эти функции (перечисленные в табл. 19.3) используют заголовок `<wchar.h>`. Заметьте, что функция `wcstok()`, в отличие от версии функции для типа `char`, требует передачи дополнительного параметра.

Таблица 19.3. Функции для операций над строками двухбайтовых символов и соответствующие им функции для типа `char`.

Функция	Соответствующая функция для типа <code>char</code>
<code>wchar_t *wcscat(wchar_t *str1, const wchar_t *str2)</code>	<code>strcat()</code> В версии C99 к параметрам <code>str1</code> и <code>str2</code> применен квалификатор <code>restrict</code>
<code>wchar_t *wcschr(const wchar_t *str, wchar_t ch)</code>	<code>strchr()</code>
<code>int wscmp(const wchar_t *str1, const wchar_t *str2)</code>	<code>strcmp()</code>
<code>int wcscoll(const wchar_t *str1, const wchar_t *str2)</code>	<code>strcoll()</code>
<code>size_t wcsncpy(const wchar_t *str1, const wchar_t *str2)</code>	<code>strncpy()</code>
<code>wchar_t *wcscpy(wchar_t *str1, const wchar_t *str2)</code>	<code>strcpy()</code> В версии C99 к параметрам <code>str1</code> и <code>str2</code> применен квалификатор <code>restrict</code>
<code>size_t wcslen(const wchar_t *str)</code>	<code>strlen()</code>
<code>wchar_t *wcsncpy(wchar_t *str1, const wchar_t *str2, size_t num)</code>	<code>strncpy()</code> В версии C99 к параметрам <code>str1</code> и <code>str2</code> применен квалификатор <code>restrict</code>
<code>wchar_t *wcsncat(wchar_t *str1, const wchar_t *str2, size_t num)</code>	<code>strncat()</code> В версии C99 к параметрам <code>str1</code> и <code>str2</code> применен квалификатор <code>restrict</code>
<code>int wcsncmp(const wchar_t *str1, const wchar_t *str2, size_t num)</code>	<code>strncmp()</code>
<code>wchar_t *wcpbrk(const wchar_t *str1, const wchar_t *str2)</code>	<code>strpbrk()</code>
<code>wchar_t *wcsrchr(const wchar_t *str1, wchar_t ch)</code>	<code>strrchr()</code>
<code>size_t wcspn(const wchar_t *str1, const wchar_t *str2)</code>	<code>strspn()</code>
<code>wchar_t *wcstok(wchar_t *str1, const wchar_t *str2, wchar_t **endptr)</code>	<code>strtok()</code> Здесь параметр <code>endptr</code> является указателем, который содержит информацию, необходимую для продолжения процесса разделения строки на лексемы. В версии C99 к параметрам <code>str1</code> и <code>str2</code> применен квалификатор <code>restrict</code>
<code>wchar_t *wcsstr(const wchar_t *str1, const wchar_t *str2)</code>	<code>strstr()</code>
<code>size_t wcsxfrm(wchar_t *str1, const wchar_t *str2, size_t num)</code>	<code>strxfrm()</code> В версии C99 к параметрам <code>str1</code> и <code>str2</code> применен квалификатор <code>restrict</code>



Преобразование строк двухбайтовых символов

Функции, перечисленные в табл. 19.4, предназначены для преобразования строк двухбайтовых символов в числа; в таблице приведены также функции времени. Для всех функций в таблице указаны соответствующие им стандартные функции. Все функции, указанные в таблице, используют заголовок `<wchar.h>`.

Таблица 19.4. Функции преобразования строк двухбайтовых символов и соответствующие им функции для типа `char`.

Функция	Соответствующая функция для типа <code>char</code>
<code>size_t wcsftime(wchar_t *str, size_t max, const wchar_t *fmt, const struct tm *ptr)</code>	<code>strftime()</code> В версии C99 к параметрам <code>str1</code> , <code>fmt</code> и <code>ptr</code> применен квалификатор <code>restrict</code>
<code>double wcstod(const wchar_t *start, wchar_t **end);</code>	<code>strtod()</code> В версии C99 к параметрам <code>start</code> и <code>end</code> применен квалификатор <code>restrict</code>
<code>float wcstof(const wchar_t * restrict start, wchar_t ** restrict end);</code>	<code>strtof()</code> (Добавлена в версии C99.)
<code>long double wcstold(const wchar_t * restrict start, wchar_t ** restrict end);</code>	<code>strtold()</code> (Добавлена в версии C99.)
<code>long int wcstol(const wchar_t *start, wchar_t **end, int radix)</code>	<code>strtol()</code> В версии C99 к параметрам <code>start</code> и <code>end</code> применен квалификатор <code>restrict</code>
<code>long long int wcstoll(const wchar_t * restrict start, wchar_t ** restrict end, int radix)</code>	<code>strtoll()</code> (Добавлена в версии C99.)
<code>unsigned long int wcstoul(const wchar_t * restrict start, wchar_t ** restrict end, int radix)</code>	<code>strtoul()</code> В версии C99 к параметрам <code>start</code> и <code>end</code> применен квалификатор <code>restrict</code>
<code>unsigned long long int wcstoull(const wchar_t *start, wchar_t **end, int radix)</code>	<code>strtoull()</code> (Добавлена в версии C99.)

Функции для обработки массивов двухбайтовых символов

Для стандартных функций, предназначенных для обработки массивов символов (например, для `memcpy()`), имеются соответствующие функции, выполняющие аналогичные операции над массивами двухбайтовых символов. Эти функции (перечисленные в следующей табл. 19.5) используют заголовок `<wchar.h>`.

Таблица 19.5. Функции для обработки массивов двухбайтовых символов и соответствующие им функции для типа `char`.

Функция	Соответствующая функция для типа <code>char</code>
<code>wchar_t *wmemchr(const wchar_t *str, wchar_t ch, size_t num)</code>	<code>memchr()</code>
<code>int wmemcmp(const wchar_t *str1, const wchar_t *str2, size_t num)</code>	<code>memcmp()</code>
<code>wchar_t *wmemcpy(wchar_t *str1, const wchar_t *str2, size_t num)</code>	<code>memcpy()</code> В версии C99 к параметрам <code>str1</code> и <code>str2</code> применен квалификатор <code>restrict</code>
<code>wchar_t *wmemmove(wchar_t *str1, const wchar_t *str2, size_t num)</code>	<code>memmove()</code>
<code>wchar_t *wmemset(wchar_t *str, wchar_t ch, size_t num)</code>	<code>memset()</code>



Функции для преобразования многобайтовых и двухбайтовых символов

Стандартная библиотека поддерживает различные функции, предназначенные для преобразования многобайтовых и двухбайтовых символов. Эти функции (перечисленные в табл. 19.6) используют заголовок `<wchar.h>`. Многие из них представляют собой версии обычных многобайтовых функций, которые могут быть прерваны (и повторно запущены — *restartable*). Повторно запускаемая версия использует информацию о состоянии, передаваемую ей в параметре типа `mbstate_t`. Если этот параметр нулевой, функция предоставит собственный объект типа `mbstate_t`.

Таблица 19.6. Функции для преобразования многобайтовых и двухбайтовых символов

Функция	Описание
<code>wint_t btowc(int ch)</code>	Преобразует параметр <i>ch</i> в его двухбайтовый эквивалент и возвращает результат. Значение <code>WEOF</code> возвращается при ошибке или если <i>ch</i> не однобайтовый, а многобайтовый символ
<code>size_t mbrlen(const char *str, size_t num, mbstate_t *state)</code>	Повторно запускаемая версия функции <code>mblen()</code> , в которой информация о состоянии передается через параметр <i>state</i> . Возвращает положительное число, равное длине следующего многобайтового символа. Ноль возвращается в случае, если следующий символ — нулевой. При ошибке возвращается отрицательное значение. В версии C99 к параметрам <i>str</i> и <i>state</i> применен квалификатор <code>restrict</code>
<code>size_t mbrtowc(wchar_t *out, const char *in, size_t num, mbstate_t *state)</code>	Повторно запускаемая версия функции <code>mbtowc()</code> , в которой информация о состоянии передается через параметр <i>state</i> . Возвращает положительное число, равное длине следующего многобайтового символа. Ноль возвращается в случае, если следующий символ — нулевой. При ошибке возвращается значение <code>-1</code> и переменной <code>errno</code> присваивается макрос <code>EILSEQ</code> . Если преобразование не завершено, возвращается число <code>-2</code> . В версии C99 к параметрам <i>out</i> , <i>in</i> и <i>state</i> применен квалификатор <code>restrict</code>
<code>int mbsinit(const mbstate_t *state)</code>	Возвращает значение <code>true</code> , если параметр <i>state</i> представляет начальное состояние процесса преобразования
<code>size_t mbsrtowcs(wchar_t *out, const char **in, size_t num, mbstate_t *state)</code>	Повторно запускаемая версия функции <code>mbstowcs()</code> , в которой информация о состоянии передается через параметр <i>state</i> . Кроме того, функция <code>mbsrtowcs()</code> отличается от функции <code>mbstowcs()</code> тем, что параметр <i>in</i> является косвенным указателем на исходный массив. При ошибке переменной <code>errno</code> присваивается макрос <code>EILSEQ</code> . В версии C99 к параметрам <i>out</i> , <i>in</i> и <i>state</i> применен квалификатор <code>restrict</code>
<code>size_t wctomb(char *out, wchar_t ch, mbstate_t *state)</code>	Повторно запускаемая версия функции <code>wctomb()</code> , в которой информация о состоянии передается через параметр <i>state</i> . При ошибке переменной <code>errno</code> присваивается макрос <code>EILSEQ</code> . В версии C99 к параметрам <i>out</i> и <i>state</i> применен квалификатор <code>restrict</code>
<code>size_t wcsrtombs(char *out, const wchar_t **in, size_t num, mbstate_t *state)</code>	Повторно запускаемая версия функции <code>wcstombs()</code> , в которой информация о состоянии передается через параметр <i>state</i> . Кроме того, функция <code>wcsrtombs()</code> отличается от функции <code>wcstombs()</code> тем, что параметр <i>in</i> является косвенным указателем на исходный массив. При ошибке переменной <code>errno</code> присваивается макрос <code>EILSEQ</code> . В версии C99 к параметрам <i>out</i> , <i>in</i> и <i>state</i> применен квалификатор <code>restrict</code>
<code>int wctob(wint_t ch)</code>	Преобразует параметр <i>ch</i> в его однобайтовый эквивалент. При сбое функция возвращает значение <code>EOF</code>

Полный
справочник по



Глава 20

**Библиотечные средства,
добавленные в версии C99**

Благодаря Стандарту C99 произошло значительное увеличение возможностей библиотеки языка C. Во-первых, добавлены новые функции в заголовки, ранее определенные в версии C89. Например, намного расширена математическая библиотека, поддерживаемая заголовком `<math.h>`. Эти дополнительные функции описывались в предыдущих главах. Во-вторых, созданы новые категории функций, начиная от поддержки арифметических операций с комплексными числами и заканчивая макросами обобщенного типа, а также новые заголовки, предназначенные для поддержки таких функций. Эти новые библиотечные элементы и описываются в данной главе.



Библиотека поддержки арифметических операций с комплексными числами

В версии C99 стало возможным выполнять арифметические операции с комплексными числами. Библиотека поддержки арифметических операций с комплексными числами имеет заголовочный файл `<complex.h>`. В нем определены следующие макросы:

Макрос	Расширение
<code>complex</code>	<code>_Complex</code>
<code>imaginary</code>	<code>_Imaginary</code>
<code>_Complex_I</code>	<code>(const float_Complex) i</code>
<code>_Imaginary_I</code>	<code>(const float_imaginary) i</code>
<code>I</code>	<code>_Imaginary_I</code> (или <code>_Complex_I</code> , если не поддерживаются мнимые типы)

Здесь *i* представляет мнимое значение, которое равно квадратному корню из -1 . Поддержка мнимых типов не обязательна.

В версии C99 вместо ключевых слов `complex` и `imaginary` определены слова `_Complex` и `_Imaginary`, поскольку во многих существующих программах, написанных на C89, уже определены пользовательские типы данных `complex` и `imaginary` для комплексных чисел. Использование в версии C99 слов `_Complex` и `_Imaginary` позволяет избежать изменения написанного ранее кода. Однако в новые программы лучше всего включить заголовок `<complex.h>`, а затем использовать макросы `complex` и `imaginary`.

На заметку

В C++ определен класс `complex`, в котором предлагается иной способ реализации действий с комплексными числами.

Ниже в табл. 20.1 приведены математические функции, часто используемые в элементарной теории функций комплексного переменного. Обратите внимание, что для каждой функции определены версии `float complex`, `double complex` и `long double complex`. Имя версии `float complex` имеет суффикс *f*, а имя версии `long double complex` — суффикс *l*. Углы измеряются, разумеется, в радианах.

Таблица 20.1. Математические функции, используемые в элементарной теории функций комплексного переменного

Функция	Назначение
<code>float cabsf(float complex arg);</code>	Возвращает абсолютную величину (модуль) комплексного числа <i>arg</i>
<code>double cabs(double complex arg);</code>	
<code>long double cabsl(long double complex arg);</code>	

Функция	Назначение
<code>float complex cacosf(float complex <i>arg</i>);</code>	Возвращает комплексное значение арккосинуса от параметра <i>arg</i>
<code>double complex cacos(double complex <i>arg</i>);</code>	
<code>long double complex cacosl(long double complex <i>arg</i>);</code>	
<code>float complex cacoshf(float complex <i>arg</i>);</code>	Возвращает комплексное значение гиперболического арккосинуса от параметра <i>arg</i>
<code>double complex cacosh(double complex <i>arg</i>);</code>	
<code>long double complex cacoshl(long double complex <i>arg</i>);</code>	
<code>float cargf(float complex <i>arg</i>);</code>	Возвращает значение аргумента комплексного числа <i>arg</i>
<code>double carg(double complex <i>arg</i>);</code>	
<code>long double cargl(long double complex <i>arg</i>);</code>	
<code>float complex casinf(float complex <i>arg</i>);</code>	Возвращает комплексное значение арксинуса от параметра <i>arg</i>
<code>double complex casin(double complex <i>arg</i>);</code>	
<code>long double complex casinl(long double complex <i>arg</i>);</code>	
<code>float complex casinhf(float complex <i>arg</i>);</code>	Возвращает комплексное значение гиперболического арксинуса от параметра <i>arg</i>
<code>double complex casinh(double complex <i>arg</i>);</code>	
<code>long double complex casinh1(long double complex <i>arg</i>);</code>	
<code>float complex catanf(float complex <i>arg</i>);</code>	Возвращает комплексное значение арктангенса от параметра <i>arg</i>
<code>double complex catan(double complex <i>arg</i>);</code>	
<code>long double complex catanl(long double complex <i>arg</i>);</code>	
<code>float complex catanhf(float complex <i>arg</i>);</code>	Возвращает комплексное значение гиперболического арктангенса от параметра <i>arg</i>
<code>double complex catanh(double complex <i>arg</i>);</code>	
<code>long double complex catanh1(long double complex <i>arg</i>);</code>	

Функция	Назначение
float complex ccosf(float complex <i>arg</i>);	Возвращает комплексное значение косинуса от параметра <i>arg</i>
double complex ccos(double complex <i>arg</i>);	
long double complex ccosl(long double complex <i>arg</i>);	
float complex ccoshf(float complex <i>arg</i>);	Возвращает комплексное значение гиперболического косинуса от параметра <i>arg</i>
double complex ccosh(double complex <i>arg</i>);	
long double complex ccoshl(long double complex <i>arg</i>);	
float complex cexpf(float complex <i>arg</i>);	Возвращает комплексное значение e^{arg} , где e — основание натурального логарифма
double complex cexp(double complex <i>arg</i>);	
long double complex cexpl(long double complex <i>arg</i>);	
float cimagf(float complex <i>arg</i>);	Возвращает мнимую часть параметра <i>arg</i>
double cimag(double complex <i>arg</i>);	
long double cimagl(long double complex <i>arg</i>);	
float complex clogf(float complex <i>arg</i>);	Возвращает комплексное значение натурального логарифма от параметра <i>arg</i>
double complex clog(double complex <i>arg</i>);	
long double complex clogl(long double complex <i>arg</i>);	
float complex conjf(float complex <i>arg</i>);	Возвращает комплексно-сопряженное значение параметра <i>arg</i>
double complex conj(double complex <i>arg</i>);	
long double complex conjl(long double complex <i>arg</i>);	
float complex cpowf(float complex <i>a</i> , long double complex <i>b</i>);	Возвращает комплексное значение a^b
double complex cpow(double complex <i>a</i> , double complex <i>b</i>);	
long double complex cpowl(long double complex <i>a</i> , long double complex <i>b</i>);	

Функция	Назначение
float complex cprojf(float complex <i>arg</i>);	Возвращает проекцию параметра <i>arg</i> на сферу Римана
double complex cproj(double complex <i>arg</i>);	
long double complex cprojl(long double complex <i>arg</i>);	
float crealf(float complex <i>arg</i>);	Возвращает вещественную часть параметра <i>arg</i>
double creal(double complex <i>arg</i>);	
long double creall(long double complex <i>arg</i>);	Возвращает комплексное значение синуса от параметра <i>arg</i>
float complex csinf(float complex <i>arg</i>);	
double complex csin(double complex <i>arg</i>);	
long double complex csinl(long double complex <i>arg</i>);	Возвращает комплексное значение гиперболического синуса от параметра <i>arg</i>
float complex csinhf(float complex <i>arg</i>);	
double complex csinh(double complex <i>arg</i>);	
long double complex csinhl(long double complex <i>arg</i>);	Возвращает комплексное значение квадратного корня из параметра <i>arg</i>
float complex csqrtf(float complex <i>arg</i>);	
double complex csqrt(double complex <i>arg</i>);	
long double complex csqrtl(long double complex <i>arg</i>);	Возвращает комплексное значение тангенса от параметра <i>arg</i>
float complex ctanf(float complex <i>arg</i>);	
double complex ctan(double complex <i>arg</i>);	
long double complex ctanl(long double complex <i>arg</i>);	Возвращает комплексное значение гиперболического тангенса от параметра <i>arg</i>
float complex ctanhf(float complex <i>arg</i>);	
double complex ctanh(double complex <i>arg</i>);	
long double complex ctanh1(long double complex <i>arg</i>);	



Библиотека поддержки среды вычислений с плавающей точкой

В версии C99 заголовком `<fenv.h>` объявляются функции, которые имеют доступ к среде вычислений с плавающей точкой. Эти функции описаны в табл. 20.2. Заголовок `<fenv.h>` также определяет типы `fenv_t` и `fexcept_t`, которые представляют конфигурацию вычислителя, реализующего среду вычислений с плавающей точкой и флаги состояния этого вычислителя соответственно. Макрос `FE_DFL_ENV` задает указатель на действующую по умолчанию конфигурацию вычислителя, реализующего среду вычислений с плавающей точкой, определенную при запуске программы.

Определены также следующие макросы исключений, возникающих при работе с числами с плавающей точкой:

<code>FE_DIVBYZERO</code>	<code>FE_INEXACT</code>	<code>FE_INVALID</code>
<code>FE_OVERFLOW</code>	<code>FE_UNDERFLOW</code>	<code>FE_ALL_EXCEPT</code>

Все комбинации этих макросов, полученные с помощью операции ИЛИ, можно сохранять в объекте типа `int`.

Определены также следующие макросы, используемые для указания направления округления значений:

<code>FE_DOWNWARD</code>	<code>FE_TONEAREST</code>	<code>FE_TOWARDZERO</code>	<code>FE_UPWARD</code>
--------------------------	---------------------------	----------------------------	------------------------

Для проверки флагов вычислителя, реализующего среду вычислений с плавающей точкой, необходимо установить специальную директиву (прагму) для компилятора `FENV_ACCESS` в положение “включено”. Разрешен ли доступ к этим флагам по умолчанию, зависит от конкретной реализации.

Таблица 20.2. Функции вычислителя, реализующего среду вычислений с плавающей точкой

Функция	Назначение
<code>void feclearexcept(int ex);</code>	Сбрасывает исключения, заданные параметром <code>ex</code>
<code>void fegetexceptflag(fexcept_t *fptr, int ex);</code>	В переменной, адресуемой указателем <code>fptr</code> , сохраняет состояние флагов исключений вычислителя, реализующего среду вычислений с плавающей точкой, заданных параметром <code>ex</code>
<code>void feraiseexcept(int ex);</code>	Возбуждает исключения, заданные параметром <code>ex</code>
<code>void fesetexceptflag(fexcept_t *fptr, int ex);</code>	Устанавливает флаги состояния вычислителя, реализующего среду вычислений с плавающей точкой, заданные параметром <code>ex</code> , в состояние флагов, содержащихся в объекте, адресуемом параметром <code>fptr</code>
<code>int fetestexcept(int ex);</code>	Выполняет операцию поразрядного ИЛИ над флагами, заданными параметром <code>ex</code> , и текущими флагами вычислителя, реализующего среду вычислений с плавающей точкой. Возвращает результат этой операции
<code>int fegetround(void);</code>	Возвращает значение действующего направления округления
<code>int fesetround(int direction);</code>	Устанавливает значение текущего направления округления с помощью параметра <code>direction</code> . При успешном выполнении возвращается ноль
<code>void fegetenv(fenv_t *envptr);</code>	В объект, адресуемый параметром <code>envptr</code> , записывается конфигурация вычислителя, реализующего среду вычислений с плавающей точкой

Функция	Назначение
<code>int feholdexcept(fenv_t *envptr);</code>	Устанавливает безостановочную обработку исключения, возникшего при выполнении вычислений с плавающей точкой. Сохраняет конфигурацию вычислителя, реализующего среду вычислений с плавающей точкой, в переменной, адресуемой параметром <i>envptr</i> , и сбрасывает флаги состояния. При успешном выполнении возвращает нуль
<code>void fesetenv(fenv_t *envptr);</code>	Устанавливает конфигурацию вычислителя, реализующего среду вычислений с плавающей точкой, равной значению переменной, адресуемой параметром <i>envptr</i> , но исключения с плавающей точкой при этом не возбуждаются. Объект, адресуемый параметром <i>envptr</i> , должен быть получен в результате вызова функции <code>fegetenv()</code> или функции <code>feholdexcept()</code>
<code>void feupdateenv(fenv_t *envptr);</code>	Устанавливает конфигурацию вычислителя, реализующего среду вычислений с плавающей точкой, равной значению переменной, адресуемой параметром <i>envptr</i> . Сначала сохраняет любые текущие исключения, а затем, после установки конфигурации вычислителя в соответствии со значением переменной, адресуемой параметром <i>envptr</i> , возбуждает эти исключения. Объект, адресуемый параметром <i>envptr</i> , должен быть получен путем вызова функции <code>fegetenv()</code> или функции <code>feholdexcept()</code>

Заголовок <stdint.h>

В заголовке `<stdint.h>` версии C99 не объявлено ни одной функции, но он определяет множество целочисленных типов и макросов. Целочисленные типы используются для объявления целых значений известного размера или значений, несущих информацию о некоторых специальных атрибутах.

Макросы, имеющие вид `intN_t`, определяют целое с разрядностью *N* бит. Например, макрос `int16_t` задает 16-разрядное целое со знаком. Макросы, имеющие вид `uintN_t`, определяют целое значение без знака с разрядностью *N* бит. Например, макрос `uint32_t` задает 32-разрядное целое без знака. Макросы, в имени которых *N* равно 8, 16, 32 или 64, доступны во всех средах, в которых предусмотрено выполнение операций над целыми числами с указанной разрядностью.

Макросы, имеющие вид `int_leastN_t`, определяют целое значение с разрядностью не менее *N* бит. Макросы, имеющие вид `uint_leastN_t`, определяют целое значение без знака с разрядностью не менее чем *N* бит. Макросы, в имени которых *N* равно 8, 16, 32 или 64, доступны во всех средах, в которых предусмотрено выполнение операций над целыми числами с указанной разрядностью. Например, макрос `int_least16_t` определяет допустимый тип.

Макросы, имеющие вид `int_fastN_t`, определяют самый быстродействующий целочисленный тип с разрядностью не менее *N* бит. Макросы, имеющие вид `uint_fastN_t`, определяют самый быстродействующий целочисленный тип без знака с разрядностью не менее чем *N* бит. Макросы, в имени которых *N* равно 8, 16, 32 или 64, доступны во всех средах. Например, макрос `int_fast32_t` — это допустимый тип значения для всех сред.

Тип `intmax_t` определяет тип целого максимальной разрядности со знаком, а тип `uintmax_t` — тип целого максимальной разрядности без знака.

Также определены типы `intptr_t` и `uintptr_t`. Их обычно используют для создания целых значений, в которых можно хранить указатели. Эти типы не являются обязательными.

В заголовке `<stdint.h>` определен ряд макросов с параметрами. Их макрорасширения являются константами заданного целочисленного типа. Эти макросы имеют следующую общую форму:

```
INTN_C(значение)
UINTN_C(значение)
```

Здесь N — разрядность нужного типа в битах. Каждый макрос создает константу разрядностью не менее N бит, которая представляет заданное значение.

Также в этом заголовке определены следующие макросы:

```
INTMAX_C(значение)
UINTMAX_C(значение)
```

Они создают константы максимальной разрядности, представляющие заданное значение.



Функции для преобразования формата целочисленных значений

В версии C99 добавлен ряд специализированных функций для преобразования формата целочисленных значений, которые позволяют преобразовывать целые значения в так называемые значения максимальной разрядности и наоборот — уменьшать разрядность при необходимости. Эти функции описаны в заголовке `<inttypes.h>`, который включает также заголовок `<stdint.h>`. Заголовок `<inttypes.h>` определяет один тип: структуру `imaxdiv_t`, в которой хранится значение, возвращаемое функцией `imaxdiv()`. Функции для преобразования формата целочисленных значений перечислены в табл. 20.3.

В заголовке `<inttypes.h>` также определено множество макросов, которые можно использовать в вызовах функций семейств `printf()` и `scanf()` для задания различных преобразований целых чисел. Макросы для функции `printf()` начинаются с префикса `PRI`, а макросы для функции `scanf()` — с префикса `SCN`. За этими префиксами стоит спецификатор преобразования, например `d` или `u`, затем следует имя типа (например, N , `MAX`, `PTR`, `FAST N` или `LEAST N` , где N задает разрядность). Точный список поддерживаемых макросов для задания различных преобразований целых чисел должен быть описан в документации к компилятору.

Таблица 20.3. Функции для преобразования целочисленных значений в формат с максимальной разрядностью и функции, выполняющие обратные преобразования

Функция	Описание
<code>intmax_t imaxabs(intmax_t arg);</code>	Возвращает абсолютное значение параметра <code>arg</code>
<code>imaxdiv_t imaxdiv(intmax_t numerator, intmax_t denominator);</code>	Возвращает структуру <code>imaxdiv_t</code> , которая содержит результат выполнения операции деления числителя <code>numerator</code> на знаменатель <code>denominator</code> . Частное занимает поле <code>quot</code> , а остаток — поле <code>rem</code> . Как поле <code>quot</code> , так и поле <code>rem</code> имеют тип <code>intmax_t</code>

Функция	Описание
<code>intmax_t strtointmax(const char * restrict start, char ** restrict end, int base);</code>	Версия функции <code>strtol()</code> для целочисленных параметров максимальной разрядности
<code>uintmax_t strtouintmax(const char * restrict start, char ** restrict end, int base);</code>	Версия функции <code>strtoul()</code> для целочисленных параметров максимальной разрядности
<code>intmax_t wcstointmax(const char * restrict start, char ** restrict end, int base);</code>	Версия функции <code>wcstol()</code> для целочисленных параметров максимальной разрядности
<code>uintmax_t wcstouintmax(const char * restrict start, char ** restrict end, int base);</code>	Версия функции <code>wcstoul()</code> для целочисленных параметров максимальной разрядности

Математические макросы обобщенного типа

Как уже было сказано в главе 15, в Стандарте C99 определены три версии для большинства математических функций — для параметров типа `float`, `double` и `long double`. Например, для вычисления синуса в стандарте C99 определены следующие функции:

```
double sin( double arg);
float sinf(float arg);
long double sinl(long double arg);
```

У всех трех функций одно и то же назначение, разница заключается лишь в типе обрабатываемых ими данных. Причем для всех функций версия, работающая с типом `double`, — это первоначальная функция, определенная в Стандарте C89, а версии для типов `float` и `long double` были добавлены в Стандарте C99. Как было отмечено в главе 15, имена функций для типа `float` имеют суффикс `f`, а имена функций для типа `long double` — суффикс `l`. (Необходимость в применении различных имен вызвана тем, что язык C не поддерживает перегрузки функций.) Предоставляя три различные функции, стандарт C99 позволяет выбрать ту из них, которая более всего приемлема в каких-то конкретных условиях. По тем же причинам каждая из математических функций комплексного аргумента также представлена тремя версиями.

Несмотря на очевидную полезность наличия трех версий математических функций и функций комплексных чисел, к сожалению, работать с ними не всегда удобно. Во-первых, при передаче данных определенного типа очень важно не забыть приписать к имени функции надлежащий суффикс. Постоянно помнить об этом довольно утомительно, и потому повышается вероятность возникновения ошибок. Во-вторых, если в процессе разработки проекта изменить тип данных, передаваемых одной из таких функций, следует изменить и суффикс в имени функции. А это, опять-таки, очень способствует “размножению” ошибок. Чтобы справиться с этими (и другими) проблемами, в Стандарте C99 определен набор макросов для обобщенного типа, которые можно использовать вместо математических или комплексных функций. Эти “универсальные” макросы автоматически транслируются в вызов нужной функции в зависимости от типа аргумента. Макросы обобщенного типа определены в заголовке `<tgmath.h>`, который автоматически включает заголовки `<math.h>` и `<complex.h>`.

Макросы обобщенного типа имеют те же имена, что и версии математических или комплексных функций для типа `double`, в вызовы которых они транслируются. (Эти имена также совпадают с именами функций, определенными в стандарте C89.) Например, макрос обобщенного типа для функций `sin()`, `sinf()` и `sinl()` использует

имя `sin()`. “Универсальный” макрос для функций `csin()`, `csinf()` и `csinl()` также имеет имя `sin()`. Как уже упоминалось, соответствующая функция вызывается в зависимости от типа аргумента. Предположим, например, что в программе определены следующие переменные:

```
long double ldbl ;
float complex fcmplx;
```

Тогда вызов

```
cos(ldbl)
```

транслируется в вызов

```
cosl(ldbl),
```

а вызов

```
cos(fcmplx)
```

транслируется в вызов

```
ccosf(fcmplx).
```

Как показано в приведенных выше примерах, макросы обобщенного типа предоставляют программисту удобное средство записи вызовов необходимых функций без потери производительности, точности или совместимости (переносимости) программного кода.

Заголовок <stdbool.h>

В стандарт C99 добавлен заголовок <stdbool.h>, который поддерживает тип данных `_Bool`. Хотя в нем не определено ни одной функции, на самом деле он определяет следующие четыре макроса.

Макрос	Расширение
<code>bool</code>	<code>_Bool</code>
<code>true</code>	<code>1</code>
<code>false</code>	<code>0</code>
<code>__bool_true_false_are_defined</code>	<code>1</code>

В версии C99 вместо ключевого слова `bool` определено ключевое слово `_Bool`, поскольку во многих существующих С-программах уже определены собственные пользовательские версии типа `bool`. Определение в версии C99 логического (булева) типа в виде ключевого слова `_Bool` позволяет избежать переписывания созданного ранее программного кода. То же объяснение относится и к ключевым словам `true` и `false`. Однако при написании новых программ лучше всего включить в них заголовок <stdbool.h>, а затем использовать макросы `bool`, `true` и `false`. Благодаря этому вы сможете создавать программы, совместимые с языком C++.

Полный справочник по



Часть IV

Алгоритмы и приложения

В части IV показано, как применить язык C для решения разнообразных задач по разработке программ. На примере разработки алгоритмов и приложений демонстрируется применение средств языка C. Многие примеры, приведенные в части IV, могут использоваться на начальном этапе при разработке собственных проектов.

Полный
справочник по



Глава 21

Сортировка и поиск

В мире компьютеров сортировка и поиск принадлежат к числу наиболее распространенных и хорошо изученных задач. Процедуры сортировки и поиска используются почти во всех программах управления базами данных, а также в компиляторах, интерпретаторах и операционных системах. В настоящей главе представлены основные алгоритмы сортировки и поиска. Как вы сможете убедиться, они также иллюстрируют некоторые важные приемы программирования на языке С. Вообще говоря, поскольку целью сортировки, является облегчение и ускорение поиска данных, алгоритмы сортировки рассматриваются в первую очередь.

Сортировка

Сортировка — это упорядочивание набора однотипных данных по возрастанию или убыванию. Сортировка является одной из наиболее приятных для умственного анализа категорией алгоритмов, поскольку процесс сортировки очень хорошо определен. Алгоритмы сортировки были подвергнуты обширному анализу, и способ их работы хорошо понятен. К сожалению, вследствие этой изученности сортировка часто воспринимается как нечто само собой разумеющееся. При необходимости отсортировать данные многие программисты просто вызывают стандартную функцию `qsort()`, входящую в стандартную библиотеку С. Однако различные подходы к сортировке обладают разными характеристиками. Несмотря на то, что некоторые способы сортировки могут быть в среднем лучше, чем другие, ни один алгоритм не является идеальным для всех случаев. Поэтому широкий набор алгоритмов сортировки — полезное добавление в инструментарий любого программиста.

Будет полезно кратко остановиться на том, почему вызов `qsort()` не является универсальным решением всех задач сортировки. Во-первых, функцию общего назначения вроде `qsort()` невозможно применить во всех ситуациях. Например, `qsort()` сортирует только массивы в памяти. Она не может сортировать данные, хранящиеся в связанных списках. Во-вторых, `qsort()` — параметризованная функция, благодаря чему она может обрабатывать широкий набор типов данных, но вместе с тем вследствие этого она работает медленнее, чем эквивалентная функция, рассчитанная на какой-то один тип данных. Наконец, как вы увидите, хотя алгоритм быстрой сортировки, примененный в функции `qsort()`, очень эффективен в общем случае, он может оказаться не самым лучшим алгоритмом в некоторых конкретных ситуациях.

Существует две общие категории алгоритмов сортировки: алгоритмы, сортирующие объекты с произвольным доступом (например, массивы или дисковые файлы произвольного доступа), и алгоритмы, сортирующие последовательные объекты (например, файлы на дисках и лентах или связанные списки¹). В данной главе рассматриваются только алгоритмы первой категории, поскольку они наиболее полезны для среднестатистического программиста.

Чаше всего при сортировке данных лишь часть их используется в качестве ключа сортировки. *Ключ* — это часть информации, определяющая порядок элементов. Таким образом, ключ участвует в сравнениях, но при обмене элементов происходит перемещение всей структуры данных. Например, в списке почтовой рассылки в качестве ключа может использоваться почтовый индекс, но сортируется весь адрес. Для простоты в нижеследующих примерах будет производиться сортировка массивов символов, в которых ключ и данные совпадают. Далее вы увидите, как адаптировать эти методы для сортировки структур данных любого типа.

¹ В зависимости от этого сортировка называется *внутренней* или *внешней*. — Прим. ред.



Классы алгоритмов сортировки

Существует три общих метода сортировки массивов¹:

- Обмен
- Выбор (выборка)
- Вставка

Чтобы понять, как работают эти методы, представьте себе колоду игральных карт. Чтобы отсортировать карты методом *обмена*¹, разложите их на столе лицом вверх и меняйте местами карты, расположенные не по порядку, пока вся колода не будет упорядочена. В методе *выбора* разложите карты на столе, выберите карту наименьшей значимости и положите ее в руку. Затем из оставшихся карт снова выберите карту наименьшей значимости и положите ее на ту, которая уже находится у вас в руке. Процесс повторяется до тех пор, пока в руке не окажутся все карты; по окончании процесса колода будет отсортирована. Чтобы отсортировать колоду методом *вставки*, возьмите все карты в руку. Выкладывайте их по одной на стол, вставляя каждую следующую карту в соответствующую позицию. Когда все карты окажутся на столе, колода будет отсортирована.



Оценка алгоритмов сортировки

Существует много различных алгоритмов сортировки. Все они имеют свои положительные стороны, но общие критерии оценки алгоритма сортировки таковы:

- Насколько быстро данный алгоритм сортирует информацию в среднем?
- Насколько быстро он работает в лучшем и худшем случаях?
- Естественно или неестественно он себя ведет?
- Переставляет ли он элементы с одинаковыми ключами?²

Давайте подробнее рассмотрим эти критерии. Очевидно, что скорость работы любого алгоритма сортировки имеет большое значение. Скорость сортировки³ массива непосредственно связана с количеством сравнений и количеством обменов, происходящих во время сортировки, причем обмены занимают больше времени. *Сравнение* происходит тогда, когда один элемент массива сравнивается с другим; *обмен* происходит тогда, когда два элемента меняются местами. Время работы одних алгоритмов сортировки растет экспоненциально, а время работы других логарифмически зависит от количества элементов.

Время работы в лучшем и худшем случаях имеет значение, если одна из этих ситуаций будет встречаться довольно часто. Алгоритм сортировки зачастую имеет хорошее среднее время выполнения, но в худшем случае он работает очень медленно.

Поведение алгоритма сортировки называется *естественным*, если время сортировки минимально для уже упорядоченного списка элементов, увеличивается по мере возрастания степени неупорядоченности списка и максимально, когда элементы списка расположены в обратном порядке. Объем работы алгоритма оценивается количеством производимых сравнений и обменов.

¹ Т.е. обменной сортировкой. — Прим. ред.

² Если в отсортированном массиве элементы с одинаковыми ключами идут в том же порядке, в котором они располагались в исходном массиве, то алгоритм сортировки называется *устойчивым*, а в противном случае — *неустойчивым*. — Прим. ред.

³ Синонимы: *быстродействие*, *эффективность*.

Чтобы понять, почему переупорядочивание элементов с одинаковыми ключами имеет определенное значение, представьте себе базу данных почтовой рассылки, упорядоченную по главному ключу и подключу. Главным ключом является почтовый индекс, а в пределах одного почтового индекса записи упорядочены по фамилии. При добавлении в список нового адреса и пересортировке списка порядок подключей (то есть фамилий внутри почтовых индексов) не должен меняться. Для гарантии, что это не произойдет, алгоритм сортировки не должен обменивать ключи с одинаковым значением¹.

Далее будут представлены характерные для каждой группы алгоритмы сортировки с анализом эффективности. После них будут продемонстрированы более совершенные методы сортировки.

Пузырьковая сортировка

Самый известный (и пользующийся дурной славой) алгоритм — *пузырьковая сортировка* (*bubble sort*, *сортировка методом пузырька*, или просто *сортировка пузырьком*)². Его популярность объясняется интересным названием и простотой самого алгоритма. Тем не менее, в общем случае это один из самых худших алгоритмов сортировки.

Пузырьковая сортировка относится к классу обменных сортировок, т.е. к классу сортировок методом обмена. Ее алгоритм содержит повторяющиеся сравнения (т.е. многократные сравнения одних и тех же элементов) и, при необходимости, обмен соседних элементов. Элементы ведут себя подобно пузырькам воздуха в воде — каждый из них поднимается на свой уровень. Простая форма алгоритма сортировки показана ниже:

```
/* Пузырьковая сортировка */
void bubble(char *items, int count)
{
    register int a, b;
    register char t;

    for(a=1; a < count; ++a)
        for(b=count-1; b >= a; --b) {
            if(items[b-1] > items[b]) {
                /* обмен элементов */
                t = items[b-1];
                items[b-1] = items[b];
                items[b] = t;
            }
        }
}
```

Здесь *items* — указатель на массив символов, подлежащий сортировке, а *count* — количество элементов в массиве. Работа пузырьковой сортировки выполняется в двух циклах. Если количество элементов массива равно *count*, внешний цикл приводит к просмотру массива *count - 1* раз. Это обеспечивает размещение элементов в правильном порядке к концу выполнения функции даже в самом худшем случае. Все сравнения и обмены выполняются во внутреннем цикле. (Слегка улучшенная версия алгоритма пузырьковой сортировки завершает работу, если при просмотре массива не было сделано ни одного обмена, но это достигается за счет добавления еще одного сравнения при каждом проходе внутреннего цикла.)

¹ Т.е. должен быть устойчивым. — *Прим. ред.*

² На самом деле есть даже два алгоритма пузырьковой сортировки: *сортировка пузырьковым включением* и *сортировка пузырьковой выборкой*. Впрочем, эффективность обоих одинакова. — *Прим. ред.*

С помощью этой версии алгоритма пузырьковой сортировки можно сортировать массивы символов по возрастанию. Например, следующая короткая программа сортирует строку, вводимую пользователем:

```
/* Программа, вызывающая функцию сортировки bubble */

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

void bubble(char *items, int count);

int main(void)
{
    char s[255];

    printf("Введите строку:");
    gets(s);
    bubble(s, strlen(s));
    printf("Отсортированная строка: %s.\n", s);

    return 0;
}
```

Чтобы наглядно показать, как работает пузырьковая сортировка, допустим, что исходный массив содержит элементы **dcab**. Ниже показано состояние массива после каждого прохода:

Начало	d c a b
Проход 1	a d c b
Проход 2	a b d c
Проход 3	a b c d

При анализе любого алгоритма сортировки полезно знать, сколько операций сравнения и обмена будет выполнено в лучшем, среднем и худшем случаях. Поскольку характеристики выполняемого кода зависят от таких факторов, как оптимизация, производимая компилятором, различия между процессорами и особенности реализации, мы не будем пытаться получить точные значения этих параметров. Вместо этого сконцентрируем свое внимание на общей эффективности каждого алгоритма.

В пузырьковой сортировке количество сравнений всегда одно и то же, поскольку два цикла `for` повторяются указанное количество раз независимо от того, был список изначально упорядочен или нет. Это значит, что алгоритм пузырьковой сортировки всегда выполняет

$$(n^2 - n) / 2$$

сравнений, где n — количество сортируемых элементов. Данная формула выведена на том основании, что внешний цикл выполняется $n - 1$ раз, а внутренний выполняется в среднем $n/2$ раз. Произведение этих величин и дает предыдущее выражение.

Обратите внимание на член n^2 в формуле. Говорят, что пузырьковая сортировка является алгоритмом *порядка* n^2 , поскольку время ее выполнения пропорционально квадрату количества сортируемых элементов. Необходимо признать, что алгоритм порядка n^2 не эффективен при большом количестве элементов, поскольку время выполнения растет экспоненциально в зависимости от количества сортируемых элементов. На рис. 21.1 показан график роста времени сортировки с увеличением размера массива.

В алгоритме пузырьковой сортировки количество обменов в лучшем случае равно нулю, если массив уже отсортирован. Однако в среднем и худшем случаях количество обменов также является величиной порядка n^2 .

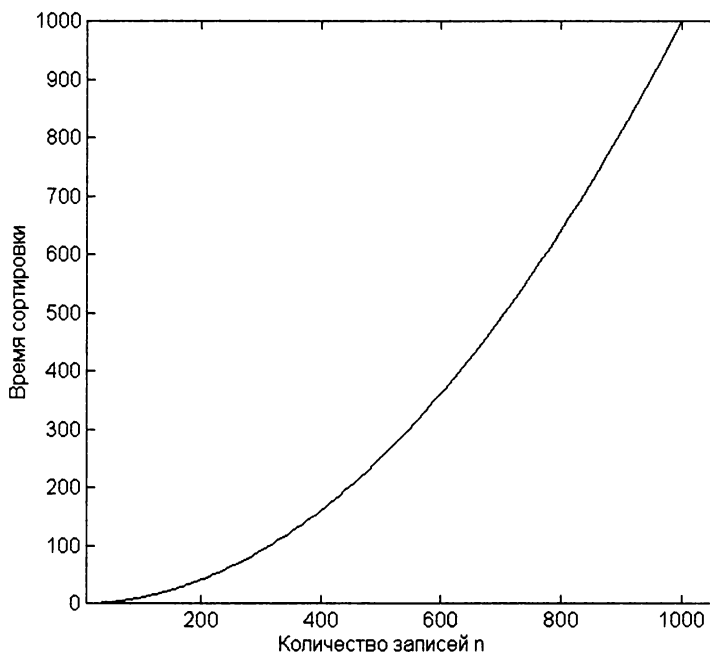


Рис. 21.1. Время сортировки порядка n^2 в зависимости от размера массива. (На самом деле здесь нарисована кривая $y = n^2/1000$, а не кривая $y = n^2$, крутизна которой в 1000 раз выше. Фактически это все равно, что нарисовать кривую $y = n^2$, выбрав по оси ординат более мелкий масштаб (в 1000 раз). Начертить кривую $y = n^2$ без растяжения вдоль оси абсцисс, на которой откладываются значения n , практически невозможно. Дело в том, что при выбранном интервале изменения n (от 0 до 1000) кривая $y = n^2$ практически сливается с осью ординат. — Прим. ред.)

Алгоритм пузырьковой сортировки можно немного улучшить, если попытаться повысить скорость его работы. Например, пузырьковая сортировка имеет такую особенность: неупорядоченные элементы на “большом” конце массива (например, “а” в примере **d c a b**) занимают правильные положения за один проход, но неупорядоченные элементы в начале массива (например, “d”) поднимаются на свои места очень медленно. Этот факт подсказывает способ улучшения алгоритма. Вместо того чтобы постоянно просматривать массив в одном направлении, в последовательных проходах можно чередовать направления. Этим мы добьемся того, что элементы, сильно удаленные от своих положений, быстро станут на свои места. Данная версия пузырьковой сортировки носит название *шейкер-сортировки* (shaker sort)¹, поскольку действия, производимые ею с массивом, напоминают взбалтывание или встряхивание. Ниже показана реализация шейкер-сортировки.

```
/* Шейкер-сортировка */
void shaker(char *items, int count)
{
    register int a;
```

¹ А также сортировки перемешиванием (cocktail shaker sort), сортировки взбалтыванием, сортировки встряхиванием. Как бы то ни было, это вид пузырьковой сортировки, в которой альтернативные проходы выполняются в противоположном направлении. — Прим. ред.

```

int exchange;
char t;

do {
    exchange = 0;
    for(a=count-1; a > 0; --a) {
        if(items[a-1] > items[a]) {
            t = items[a-1];
            items[a-1] = items[a];
            items[a] = t;
            exchange = 1;
        }
    }

    for(a=1; a < count; ++a) {
        if(items[a-1] > items[a]) {
            t = items[a-1];
            items[a-1] = items[a];
            items[a] = t;
            exchange = 1;
        }
    }
} while(exchange); /* сортировать до тех пор, пока не будет обменов */
}

```

Хотя шейкер-сортировка и является улучшенным вариантом по сравнению с пузырьковой сортировкой, она по-прежнему имеет время выполнения порядка n^2 . Это объясняется тем, что количество сравнений не изменилось, а количество обменов уменьшилось лишь на относительно небольшую константу. Шейкер-сортировка лучше пузырьковой, но есть еще гораздо лучшие алгоритмы сортировки.

Сортировка посредством выбора

При сортировке посредством выбора¹ из массива выбирается элемент с наименьшим значением и обменивается с первым элементом. Затем из оставшихся $n - 1$ элементов снова выбирается элемент с наименьшим ключом и обменивается со вторым элементом, и т. д. Эти обмены продолжаются до двух последних элементов. Например, если применить метод выбора к массиву **dcab**, каждый проход будет выглядеть так, как показано ниже:

Начало	d c a b
Проход 1	a c d b
Проход 2	a b d c
Проход 3	a b c d

Нижеследующий код демонстрирует простейшую сортировку посредством выбора:

```

/* Сортировка посредством выбора. */
void select(char *items, int count)
{
    register int a, b, c;
    int exchange;
    char t;

    for(a=0; a < count-1; ++a) {
        exchange = 0;

```

¹ Называется также *сортировкой выбором* и *сортировкой выборками*. — Прим. ред.

```

    c = a;
    t = items[a];
    for(b=a+1; b < count; ++b) {
        if(items[b] < t) {
            c = b;
            t = items[b];
            exchange = 1;
        }
    }
    if(exchange) {
        items[c] = items[a];
        items[a] = t;
    }
}

```

К сожалению, как и в пузырьковой сортировке, внешний цикл выполняется $n - 1$ раз, а внутренний — в среднем $n/2$ раз. Следовательно, сортировка посредством выбора требует $1/2 (n^2 - n)$

сравнений. Таким образом, это алгоритм порядка n^2 , из-за чего он считается слишком медленным для сортировки большого количества элементов. Несмотря на то, что количество сравнений в пузырьковой сортировке и сортировке посредством выбора одинаковое, в последней количество обменов в среднем случае намного меньше, чем в пузырьковой сортировке.



Сортировка вставками

Сортировка вставками — третий и последний из простых алгоритмов сортировки. Сначала он сортирует два первых элемента массива. Затем алгоритм вставляет третий элемент в соответствующую порядку позицию по отношению к первым двум элементам. После этого он вставляет четвертый элемент в список из трех элементов. Этот процесс повторяется до тех пор, пока не будут вставлены все элементы. Например, при сортировке массива **dcab** каждый проход алгоритма будет выглядеть следующим образом:

Начало	d c a b
Проход 1	c d a b
Проход 2	a c d b
Проход 3	a b c d

Пример реализации сортировки вставками показан ниже:

```

/* Сортировка вставками. */
void insert(char *items, int count)
{
    register int a, b;
    char t;

    for(a=1; a < count; ++a) {
        t = items[a];
        for(b=a-1; (b >= 0) && (t < items[b]); b--)
            items[b+1] = items[b];
        items[b+1] = t;
    }
}

```


В отличие от пузырьковой сортировки и сортировки посредством выбора, количество сравнений в сортировке вставками зависит от изначальной упорядоченности списка. Если список уже отсортирован, количество сравнений равно $n - 1$; в противном случае его производительность является величиной порядка n^2 .

Вообще говоря, в худших случаях сортировка вставками настолько же плоха, как и пузырьковая сортировка и сортировка посредством выбора, а в среднем она лишь немного лучше. Тем не менее, у сортировки вставками есть два преимущества. Во-первых, ее поведение естественно. Другими словами, она работает меньше всего, когда массив уже упорядочен, и больше всего, когда массив отсортирован в обратном порядке. Поэтому сортировка вставками — идеальный алгоритм для почти упорядоченных списков. Второе преимущество заключается в том, что данный алгоритм не меняет порядок одинаковых ключей¹. Это значит, что если список отсортирован по двум ключам, то после сортировки вставками он останется упорядоченным по обоим.

Несмотря на то, что количество сравнений при определенных наборах данных может быть довольно низким, при каждой вставке элемента на свое место массив необходимо сдвигать. Вследствие этого количество перемещений может быть значительным.



Улучшенные алгоритмы сортировки

Все алгоритмы, рассмотренные в предыдущих разделах, имеют один фатальный недостаток — время их выполнения имеет порядок n^2 . Это делает сортировку больших объемов данных очень медленной. По существу, в какой-то момент эти алгоритмы становятся слишком медленными, чтобы их применять². К сожалению, страшные истории о “сортировках, которые продолжались три дня”, зачастую реальны. Когда сортировка занимает слишком много времени, причиной этому обычно является неэффективность использованного в ней алгоритма. Тем не менее, первой реакцией в такой ситуации часто становится оптимизация кода вручную, возможно, путем переписывания его на ассемблере. Несмотря на то, что ручная оптимизация иногда ускоряет процедуру на постоянный множитель³, если алгоритм сортировки не эффективен, сортировка всегда будет медленной независимо от того, насколько оптимально

¹ Т.е. устойчив. — *Прим. ред.*

² Конечно, это не означает, что функция $f(n)=n^2$ в какой-то точке возрастает скачкообразно. Вовсе нет! Просто при увеличении размера массива n меняется характер сортировки, из *внутренней* она *фактически* становится *внешней*, когда массив не помещается в оперативной памяти и начинается интенсивная подкачка страниц, а за ней пробуксовывание механизма виртуальной памяти. Вот эти-то события действительно могут наступить внезапно, и тогда может показаться, что незначительное увеличение сортируемого массива или просто добавление какой-либо совершенно незначительной задачи приведет к катастрофическому увеличению времени сортировки (например в десятки раз!). — *Прим. ред.*

³ Отдельные программисты — “любители рассказов о рыбной ловле” — клянутся об увеличении эффективности сначала наполовину, затем вдвое-втрое, к середине рассказа — на порядок, а к концу рассказа — на несколько порядков. (Такое не получается даже в специально подобранных примерах для рекламного проспекта по языку Ассемблера.) На самом деле производительность может даже упасть. В лучшем случае удается повысить ее на 10-12% для реально значимых производственных задач. При этом чем сложнее алгоритм, тем сложнее переписать его на Ассемблере и тем проще сделать в нем ошибку при переписывании, а тем более сложнее ее найти. Кроме того, следует учитывать и такой фактор: например, программу писал какой-то квалифицированный программист, который выбрал простой (но не очень эффективный — причем об этом он знал) алгоритм потому, что менеджеры настаивали на скорейшем завершении программы, а оптимизацию этой программы те же менеджеры поручат весьма не самым квалифицированным специалистам! Эффект действительно будет на несколько порядков больше, но в совершенно противоположную сторону! Ведь с таким же успехом для “улучшения” трагедий Шекспира за пишущие машинки можно было усадить стадо обезьян! — *Прим. ред.*

написан код. Следует помнить, если время работы процедуры пропорционально n^2 , то увеличение скорости кода или компьютера даст лишь небольшое улучшение¹, поскольку время выполнения увеличивается как n^2 . (На самом деле, кривая n^2 на рис. 21.1 растянута вправо, но в остальном соответствует действительности.) Существует правило: если используемый в программе алгоритм слишком медленный сам по себе, никакой объем ручной оптимизации не сделает программу достаточно быстрой. Решение заключается в применении лучшего алгоритма сортировки.

Ниже описаны два прекрасных метода сортировки. Первый называется *сортировкой Шелла*. Второй — *быстрая сортировка* — обычно считается самым лучшим алгоритмом сортировки. Оба метода являются более совершенными способами сортировки и имеют намного лучшую общую производительность, чем любой из приведенных выше простых методов.

Сортировка Шелла

Сортировка Шелла называется так по имени своего автора, Дональда Л. Шелла (Donald Lewis Shell)². Однако это название закрепилось, вероятно, также потому, что действие этого метода часто иллюстрируется рядами морских раковин, перекрывающих друг друга (по-английски “shell” — “раковина”). Общая идея заимствована из сортировки вставками и основывается на уменьшении шагов³. Рассмотрим диаграмму на рис. 21.2. Сначала сортируются все элементы, отстоящие друг от друга на три позиции. Затем сортируются элементы, расположенные на расстоянии двух позиций. Наконец, сортируются все соседние элементы.

То, что этот метод дает хорошие результаты, или даже то, что он вообще сортирует массив, увидеть не так просто. Тем не менее, это верно. Каждый проход сортировки распространяется на относительно небольшое количество элементов либо на элементы, расположенные уже в относительном порядке. Поэтому сортировка Шелла эффективна, а каждый проход повышает упорядоченность⁴.

¹ Действительно, чтобы увеличить в m раз размер сортируемого массива при сохранении времени сортировки, быстродействие процессора придется увеличить в m^2 раз при условии, что время доступа к элементам массива не увеличится, т.е. не уменьшится, например, эффективность подкачки страниц. — *Прим. ред.*

² Считается, что Дональд Л. Шелл описал свой метод сортировки 28 июля 1959 года. Данный метод классифицируется как *слияние с обменом*; часто называется также *сортировкой с убывающим шагом*. — *Прим. ред.*

³ Шаг — расстояние между сортируемыми элементами на конкретном этапе сортировки. — *Прим. ред.*

⁴ Т.е. уменьшает количество беспорядков (инверсий). — *Прим. ред.*

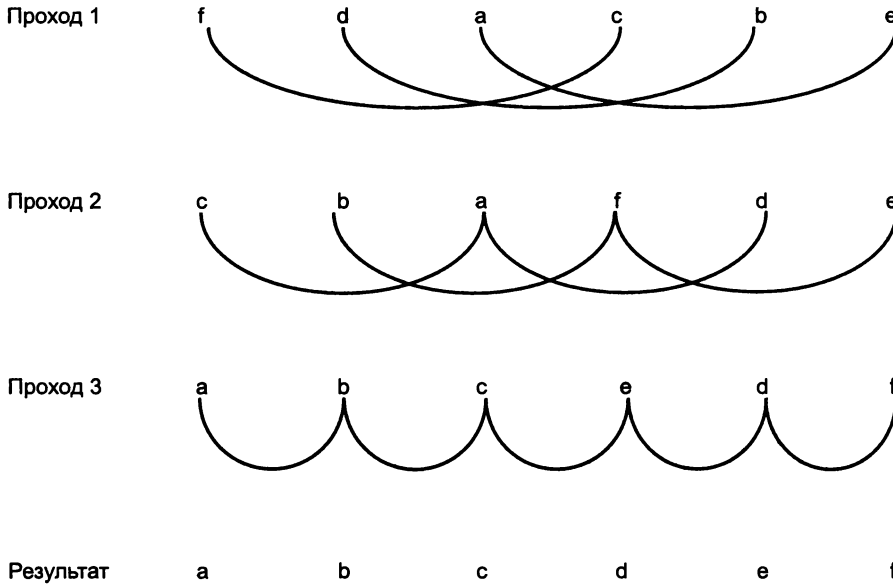


Рис. 21.2. Сортировка Шелла

Конкретная последовательность шагов может быть и другой. Единственное правило состоит в том, чтобы последний шаг был равен 1. Например, такая последовательность:

9, 5, 3, 2, 1

дает хорошие результаты и применяется в показанной здесь реализации сортировки Шелла. Следует избегать последовательностей, которые являются степенями числа 2 — по математически сложным соображениям они уменьшают эффективность сортировки (но сортировка по-прежнему работает!).

```
/* Сортировка Шелла. */
void shell(char *items, int count)
{
    register int i, j, gap, k;
    char x, a[5];

    a[0]=9; a[1]=5; a[2]=3; a[3]=2; a[4]=1;

    for(k=0; k < 5; k++) {
        gap = a[k];
        for(i=gap; i < count; ++i) {
            x = items[i];
            for(j=i-gap; (x < items[j]) && (j >= 0); j=j-gap)
                items[j+gap] = items[j];
            items[j+gap] = x;
        }
    }
}
```

Вы могли заметить, что внутренний цикл `for` имеет два условия проверки. Очевидно, что сравнение `x < items[j]` необходимо для процесса сортировки. Выражение `j >= 0` предотвращает выход за границу массива `items`. Эти дополнительные проверки в некоторой степени понижают производительность сортировки Шелла.

В слегка модифицированных версиях данного метода сортировки применяются специальные элементы массива, называемые *сигнальными метками*. Они не принадлежат к собственно сортируемому массиву, а содержат специальные значения, соответствующие наименьшему возможному и наибольшему возможному элементам¹. Это устраняет необходимость проверки выхода за границы массива. Однако применение сигнальных меток элементов требует конкретной информации о сортируемых данных, что уменьшает универсальность функции сортировки.

Анализ сортировки Шелла связан с очень сложными математическими задачами, которые выходят далеко за рамки этой книги. Примите на веру, что время сортировки пропорционально

$$n^{1,2}$$

при сортировке n элементов². А это уже существенное улучшение по сравнению с сортировками порядка n^2 . Чтобы понять, насколько оно велико, обратитесь к рис. 21.3, на котором показаны графики функций n^2 и $n^{1,2}$. Тем не менее, не стоит чрезмерно восхищаться сортировкой Шелла — быстрая сортировка еще лучше.

Быстрая сортировка

Быстрая сортировка, придуманная Ч. А. Р. Хоаром³ (Charles Antony Richard Hoare) и названная его именем, является самым лучшим методом сортировки из представленных в данной книге и обычно считается лучшим из существующих в настоящее время алгоритмом сортировки общего назначения. В ее основе лежит сортировка обменами — удивительный факт, учитывая ужасную производительность пузырьковой сортировки!

Быстрая сортировка построена на идее деления. Общая процедура заключается в том, чтобы выбрать некоторое значение, называемое *компарандом* (comparand)⁴, а затем разбить массив на две части. Все элементы, большие или равные компаранду, перемещаются на одну сторону, а меньшие — на другую. Потом этот процесс повторяется для каждой части до тех пор, пока массив не будет отсортирован. Например, если исходный массив состоит из символов **fedacb**, а в качестве компаранда используется символ **d**, первый проход быстрой сортировки перепорядочит массив следующим образом:

Начало	f e d a c b
Проход 1	b c a d e f

¹ $-\infty$ и $+\infty$. — Прим. ред.

² Вообще говоря, время сортировки Шелла зависит от последовательности шагов. (Впрочем, минимум равен, конечно, $n \log_2 n$.) Оптимальная последовательность не известна до сих пор. Дональд Кнут исследовал различные последовательности (не забыв и последовательность Фибоначчи). Фактически он пришел к выводу, что в определении наилучшей последовательности есть какое-то “колдовство”. В 1969 г. Воган Пратт обнаружил, что *если все шаги выбираются из множества чисел вида 2^{p3^q} , меньших n , то время работы будет порядка $n(\log n)^2$* . А.А. Папернов и Г.В. Стасевич в 1965 г. доказали, что максимальное время сортировки Шелла не превосходит $O(n^{1,5})$, причем уменьшить показатель 1.5 нельзя. Большое число экспериментов с сортировкой Шелла провели Джеймс Петерсон и Дэвид Л. Рассел в Стэнфордском университете в 1971 г. Они пытались определить среднее число перемещений при $100 \leq n \leq 250\,000$ для последовательности шагов 2^k-1 . Наиболее подходящими формулами оказались $1.21n^{1,26}$ и $.39n(\ln n)-2.33n \ln n$. Но при изменении диапазона n оказалось, что коэффициенты в представлении степенной функцией практически не изменяются, а коэффициенты в логарифмическом представлении изменяются довольно резко. Поэтому естественно предположить, что именно степенная функция описывает истинное асимптотическое поведение сортировки Шелла. — Прим. ред.

³ Встречается также написание Ч. Э. Р. Хоор. — Прим. ред.

⁴ *Компаранд* — операнд в операции сравнения. Иногда называется также *основой* и *критерием разбиения*. — Прим. ред.

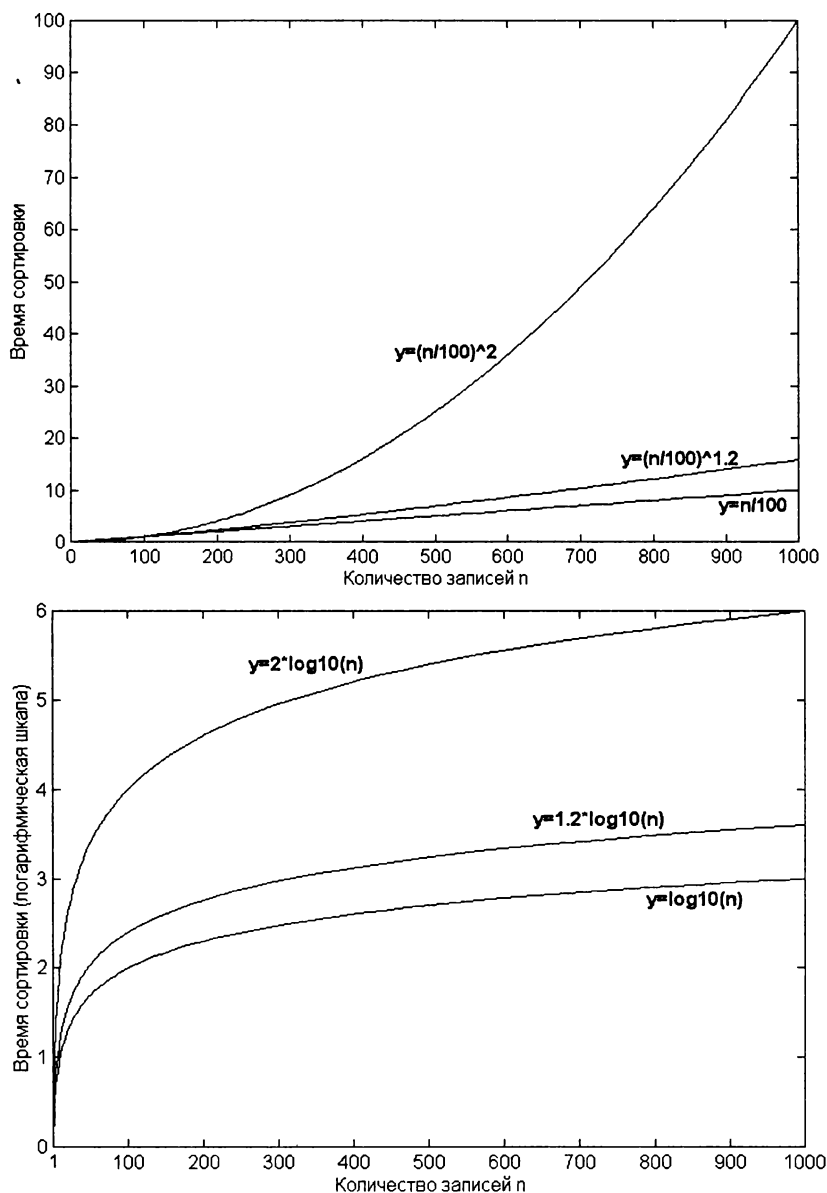


Рис. 21.3. Попытка наглядного представления кривых n^2 и $n^{1.2}$. Хотя вычертить эти кривые с точным соблюдением масштаба на каком-нибудь значимом для целей сортировки интервале изменения количества записей (n), например, на интервале от 0 до 1000, не представляется возможным, получить представление о поведении этих кривых можно с помощью графиков функций $y=(n/100)^2$ и $y=(n/100)^{1.2}$. Для сравнения построен также график прямой $y=n/100$. Кроме того, чтобы получить представление о росте этих кривых, можно на оси ординат принять логарифмический масштаб, — это все равно, что начертить логарифмы этих функций

Затем сортировка повторяется для обеих половин массива, то есть **bca** и **def**. Как вы видите, этот процесс по своей сути рекурсивный, и, действительно, в чистом виде быстрая сортировка реализуется как рекурсивная функция¹.

Значение компаранда можно выбирать двумя способами — случайным образом либо усреднив небольшое количество значений из массива. Для оптимальной сортировки необходимо выбирать значение, которое расположено точно в середине диапазона всех значений. Однако для большинства наборов данных это сделать непросто. В худшем случае выбранное значение оказывается одним из крайних. Тем не менее, даже в этом случае быстрая сортировка работает правильно. В приведенной ниже версии быстрой сортировки в качестве компаранда выбирается средний элемент массива.

```
/* Функция, вызывающая функцию быстрой сортировки. */
void quick(char *items, int count)
{
    qs(items, 0, count-1);
}

/* Быстрая сортировка. */
void qs(char *items, int left, int right)
{
    register int i, j;
    char x, y;

    i = left; j = right;
    x = items[(left+right)/2]; /* выбор компаранда */

    do {
        while((items[i] < x) && (i < right)) i++;
        while((x < items[j]) && (j > left)) j--;

        if(i <= j) {
            y = items[i];
            items[i] = items[j];
            items[j] = y;
            i++; j--;
        }
    } while(i <= j);

    if(left < j) qs(items, left, j);
    if(i < right) qs(items, i, right);
}
```

В этой версии функция `quick()` готовит вызов главной сортирующей функции `qs()`. Это обеспечивает общий интерфейс с параметрами `items` и `count`, но несущественно, так как можно вызывать непосредственно функцию `qs()` с тремя аргументами.

Получение количества сравнений и обменов, которые выполняются при быстрой сортировке, требует математических выкладок, которые выходят за рамки данной книги. Тем не менее, среднее количество сравнений равно

$$n \log n$$

а среднее количество обменов примерно равно

$$n/6 \log n$$

¹ Если хотите избежать рекурсии, не волнуйтесь, все очень легко переписывается даже для Фортрана IV, в упомянутой ранее литературе вы без труда найдете нужный не рекурсивный вариант. — *Прим. ред.*

Эти величины намного меньше соответствующих характеристик рассмотренных ранее алгоритмов сортировки.

Необходимо упомянуть об одном особенно проблематичном аспекте быстрой сортировки. Если значение компаранда в каждом делении равно наибольшему значению, быстрая сортировка становится “медленной сортировкой” со временем выполнения порядка n^2 . Поэтому внимательно выбирайте метод определения компаранда. Этот метод часто определяется природой сортируемых данных. Например, в очень больших списках почтовой рассылки, в которых сортировка происходит по почтовому индексу, выбор прост, потому что почтовые индексы довольно равномерно распределены — компаранд можно определить с помощью простой алгебраической функции. Однако в других базах данных зачастую лучшим выбором является случайное значение. Популярный и довольно эффективный метод — выбрать три элемента из сортируемой части массива и взять в качестве компаранда значение, расположенное между двумя другими.



Выбор метода сортировки

Каждый программист должен располагать широким набором алгоритмов сортировки. Несмотря на то, что в среднем случае оптимальной является именно быстрая сортировка, она не является лучшей во всех случаях. Например, при сортировке очень маленьких списков (например, менее 100 элементов) дополнительный объем работы, создаваемый рекурсивными вызовами быстрой сортировки, может перекрыть преимущества ее более хорошего алгоритма. В таких редких случаях один из простых методов сортировки — возможно, даже пузырьковая сортировка — может работать быстрее. Кроме того, если известно, что список уже почти упорядочен или если вы не хотите переставлять одинаковые ключи, какой-либо другой алгоритм подойдет лучше, чем быстрая сортировка. Суть сказанного заключается в том, что лишь тот факт, что быстрая сортировка является лучшим алгоритмом общего назначения, не означает, что в конкретных случаях другие подходы не дадут лучших результатов.



Сортировка других структур данных

До сих пор мы сортировали только массивы символов. Очевидно, что приведенные выше функции можно переделать для сортировки массивов любого из встроенных типов данных, просто поменяв типы параметров и переменных. Тем не менее, обычно возникает необходимость сортировать составные типы данных, например строки, или агрегированные данные, например структуры. Большинство задач сортировки имеют дело с ключом и информацией, связанной с этим ключом. Чтобы адаптировать алгоритмы для обработки подобных данных, необходимо модифицировать код сравнения, код обмена или оба фрагмента. Сам алгоритм при этом не меняется.

Поскольку быстрая сортировка в настоящее время является одним из лучших методов сортировки общего назначения, она используется в последующих примерах. Тем не менее, тот же принцип относится и ко всем остальным методам, описанным ранее.

Сортировка строк

Сортировка строк является распространенной задачей программирования. Строки легче всего сортировать, когда они хранятся в таблице строк. Таблица строк — это просто массив строк. А массив строк — это двумерный массив символов, в котором количество строк в таблице определяется размером левого измерения, а максимальная длина строки — размером правого измерения. (О массивах строк рассказывалось в

главе 4.) Нижеследующая строковая версия быстрой сортировки принимает массив строк, в котором размер каждой строки ограничен десятью символами. (Можете изменить эту длину, если хотите.) Данная версия сортирует строки в лексикографическом порядке.

```
/* Быстрая сортировка строк. */
void quick_string(char items[][10], int count)
{
    qs_string(items, 0, count-1);
}

void qs_string(char items[][10], int left, int right)
{
    register int i, j;
    char *x;
    char temp[10];

    i = left; j = right;
    x = items[(left+right)/2];

    do {
        while((strcmp(items[i],x) < 0) && (i < right)) i++;
        while((strcmp(items[j],x) > 0) && (j > left)) j--;
        if(i <= j) {
            strcpy(temp, items[i]);
            strcpy(items[i], items[j]);
            strcpy(items[j], temp);
            i++; j--;
        }
    } while(i <= j);

    if(left < j) qs_string(items, left, j);
    if(i < right) qs_string(items, i, right);
}
```

Обратите внимание, что во фрагменте сравнения теперь используется функция `strcmp()`. Эта функция возвращает отрицательное число, если первая строка лексикографически меньше второй, возвращает ноль, если строки равны, и положительное число, если первая строка лексикографически больше второй. Также следует отметить, что для обмена двух строк требуется три вызова функции `strcpy()`.

Имейте в виду, что функция `strcmp()` замедляет сортировку по двум причинам. Во-первых, в программе появляется вызов функции, что всегда отнимает время. Во-вторых, сама функция `strcmp()` выполняет несколько сравнений, чтобы определить, какая из двух строк больше. В первом случае, если скорость очень важна, можно поместить код сравнения строк непосредственно в функцию сортировки, продублировав код функции `strcmp()`. Во втором случае нет никакого способа избежать сравнения строк, поскольку по определению это именно то, что требуется в данной задаче. Те же рассуждения относятся и к функции `strcpy()`. Обмен двух строк с помощью `strcpy()` включает в себя вызов функции и посимвольный обмен содержимого строк — каждая из этих операций занимает время. Накладные расходы на вызов функции можно устранить, вставив код копирования прямо в алгоритм сортировки. Однако тот факт, что обмен двух строк означает обмен отдельных символов (один за другим), изменить невозможно.

Ниже приведена простая функция `main()`, демонстрирующая работу функции быстрой сортировки строк `quick_string()`:


```

#include <stdio.h>
#include <string.h>

void quick_string(char items[][10], int count);
void qs_string(char items[][10], int left, int right);

char str[][10] = { "один",
                  "два",
                  "три",
                  "четыре"
                };

int main(void)
{
    int i;

    quick_string(str, 4);

    for(i=0; i<4; i++) printf("%s ", str[i]);

    return 0;
}

```

Сортировка структур

В большинстве прикладных программ, в которых используется сортировка, предусмотрена сортировка совокупностей данных. Например, списки почтовой рассылки, складские базы данных и журналы сотрудников содержат наборы разнотипных данных. Как вам известно, в программах на языке C совокупности данных обычно хранятся в структурах. Хотя структура обычно содержит несколько членов, структуры, как правило, сортируются только по одному полю-члену, который используется в качестве ключа сортировки. За исключением выбора ключа, приемы сортировки структур ничем не отличаются от приемов сортировки других типов данных.

Чтобы проиллюстрировать пример сортировки структур, давайте создадим структуру под названием `address`, в которой можно хранить почтовый адрес. Подобная структура может применяться в программе почтовой рассылки. Описание структуры `address` показано ниже:

```

struct address {
    char name[40];    /* имя */
    char street[40]; /* улица */
    char city[20];    /* город */
    char state[3];    /* штат */
    char zip[11];     /* индекс */
};

```

Поскольку представляется разумным организовать список адресов в виде массива структур, в данном примере предположим, что процедура сортировки будет сортировать массив структур типа `address`. Такая процедура показана ниже. Она сортирует адреса по почтовому индексу.

```

/* Быстрая сортировка структур типа address. */
void quick_struct(struct address items[], int count)
{
    qs_struct(items, 0, count-1);
}

void qs_struct(struct address items[], int left, int right)

```

```

{
    register int i, j;
    char *x;
    struct address temp;

    i = left; j = right;
    x = items[(left+right)/2].zip; /* сортировка по почтовому
                                   индексу */

    do {
        while((strcmp(items[i].zip,x) < 0) && (i < right)) i++;
        while(!(strcmp(items[j].zip,x) > 0) && (j > left)) j--;
        if(i <= j) {
            temp = items[i];
            items[i] = items[j];
            items[j] = temp;
            i++; j--;
        }
    } while(i <= j);

    if(left < j) qs_struct(items, left, j);
    if(i < right) qs_struct(items, i, right);
}

```



Сортировка дисковых файлов с произвольной выборкой

Дисковые файлы бывают двух типов: с *последовательной выборкой (доступом)* и с *произвольной выборкой*. Если дисковый файл любого типа достаточно мал, его можно считать в память и отсортировать одной из процедур сортировки массивов, представленных выше. Однако многие дисковые файлы слишком велики для того, чтобы сортировать их в память, а поэтому требуют особенных приемов сортировки. Многие приложения баз данных работают с файлами с произвольной выборкой. В данном разделе показан один способ сортировки таких файлов.

Дисковые файлы с произвольной выборкой доступа имеют два больших преимущества перед файлами с последовательной выборкой. Во-первых, с ними легко работать. Для обновления информации не нужно копировать весь список. Во-вторых, их можно рассматривать как очень большой массив на диске, что значительно упрощает сортировку.

Тот факт, что файл с произвольной выборкой можно рассматривать как массив, означает, что к нему можно применить быструю сортировку лишь с небольшим количеством изменений. Вместо обращения к элементам массива по индексу, дисковая версия быстрой сортировки должна пользоваться функцией `fseek()`, чтобы найти соответствующую запись на диске.

В каждой реальной ситуации сортировка определяется конкретной структурой сортируемых данных и ключом сортировки. Тем не менее, общую идею сортировки дисковых файлов с произвольной выборкой можно понять на примере короткой программы, сортирующей структуры типа `address` — почтовые структуры, описанные ранее. Эта программа, приведенная ниже, сначала создает дисковый файл, содержащий неупорядоченные адреса. Затем она сортирует этот файл. Количество адресов, подлежащих сортировке, указано константой `NUM_ELEMENTS` (которая равна 4 в данной программе). Однако в реальной жизни количество записей придется отслеживать

динамически. Поэкспериментируйте с этой программой самостоятельно, пробуя различные типы структур, содержащие разнообразные данные:

```
/* Дисксовая сортировка структур типа address. */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define NUM_ELEMENTS 4 /* Количество элементов — произвольное
                        число, которое должно определяться
                        динамически для каждого списка. */

struct address {
    char name[30];
    char street[40];
    char city[20];
    char state[3];
    char zip[11];
} ainfo;

struct address addrs[NUM_ELEMENTS] = {
    "A. Alexander", "101 1st St", "Olney", "Ga", "55555",
    "B. Bertrand", "22 2nd Ave", "Oakland", "Pa", "34232",
    "C. Carlisle", "33 3rd Blvd", "Ava", "Or", "92000",
    "D. Dodger", "4 Fourth Dr", "Fresno", "Mi", "45678"
};

void quick_disk(FILE *fp, int count);
void qs_disk(FILE *fp, int left, int right);
void swap_all_fields(FILE *fp, long i, long j);
char *get_zip(FILE *fp, long rec);

int main(void)
{
    FILE *fp;

    /* сначала создадим файл с данными, подлежащий сортировке */
    if((fp=fopen("mlist", "wb"))==NULL) {
        printf("Невозможно открыть файл на запись.\n");
        exit(1);
    }
    printf("Запись неупорядоченных данных в дисковый файл.\n");
    fwrite(addrs, sizeof(addrs), 1, fp);
    fclose(fp);

    /* теперь отсортируем файл */
    if((fp=fopen("mlist", "rb+"))==NULL) {
        printf("Невозможно открыть файл на чтение/запись.\n");
        exit(1);
    }

    printf("Сортировка дискового файла.\n");
    quick_disk(fp, NUM_ELEMENTS);
    fclose(fp);

    printf("Файл отсортирован.\n");

    return 0;
}
```

```

}

/* Быстрая сортировка файлов. */
void quick_disk(FILE *fp, int count)
{
    qs_disk(fp, 0, count-1);
}

void qs_disk(FILE *fp, int left, int right)
{
    long int i, j;
    char x[100];

    i = left; j = right;

    strcpy(x, get_zip(fp, (long)(i+j)/2)); /* получить средний почтовый
                                             индекс */

    do {
        while((strcmp(get_zip(fp,i),x) < 0) && (i < right)) i++;
        while((strcmp(get_zip(fp,j),x) > 0) && (j > left)) j--;

        if(i <= j) {
            swap_all_fields(fp, i, j);
            i++; j--;
        }
    } while(i <= j);

    if(left < j) qs_disk(fp, left, (int) j);
    if(i < right) qs_disk(fp, (int) i, right);
}

void swap_all_fields(FILE *fp, long i, long j)
{
    char a[sizeof(ainfo)], b[sizeof(ainfo)];

    /* считать в память записи i и j */
    fseek(fp, sizeof(ainfo)*i, SEEK_SET);
    fread(a, sizeof(ainfo), 1, fp);

    fseek(fp, sizeof(ainfo)*j, SEEK_SET);
    fread(b, sizeof(ainfo), 1, fp);

    /* потом записать их на диск, поменяв местами */
    fseek(fp, sizeof(ainfo)*j, SEEK_SET);
    fwrite(a, sizeof(ainfo), 1, fp);
    fseek(fp, sizeof(ainfo)*i, SEEK_SET);
    fwrite(b, sizeof(ainfo), 1, fp);
}

/* Получение указателя на почтовый код записи */
char *get_zip(FILE *fp, long rec)
{
    struct address *p;

    p = &ainfo;

    fseek(fp, rec*sizeof(ainfo), SEEK_SET);

```

```

fread(p, sizeof(ainfo), 1, fp);

return ainfo.zip;
}

```

Как вы, наверное, уже заметили, для сортировки адресных записей пришлось написать две вспомогательные функции. Во фрагменте сравнения используется функция `get_zip()`, возвращающая указатель на поле `zip` сравниваемых записей. Функция `swap_all_fields()` выполняет обмен данных двух записей. Порядок операций чтения и записи оказывает большое влияние на скорость выполнения сортировки. При обмене двух записей программа перемещает указатель текущей записи в файле сначала на запись `i`, а потом на запись `j`. Пока головка диска находится над записью `j`, записываются данные записи `j`. Это значит, что в этот момент головку не придется перемещать на большое расстояние. Если бы код был составлен так, что первой записывалась бы запись `i`, понадобилось бы еще одно перемещение головки.

Поиск

Базы данных существуют для того, чтобы время от времени пользователи могли найти нужную запись, введя ее ключ. Существует один метод поиска информации в неупорядоченном массиве, и другой для поиска в упорядоченном массиве. В набор стандартной библиотеки компиляторов языка C входит стандартная функция `bsearch()`. Тем не менее, как и в случае сортировки, процедуры общего назначения иногда совсем не эффективны при использовании в критических ситуациях из-за накладных расходов, связанных с их обобщением. Кроме того, функцию `bsearch()` невозможно применить к неупорядоченным данным.

Методы поиска

Для нахождения информации в неупорядоченном массиве требуется последовательный поиск, начинающийся с первого элемента и заканчивающийся при обнаружении подходящих данных либо при достижении конца массива. Этот метод применим для неупорядоченной информации, но также можно использовать его и на отсортированных данных. Однако если данные уже отсортированы, можно применить двоичный поиск, который находит данные быстрее.

Последовательный поиск

Последовательный поиск очень легко запрограммировать. Приведенная ниже функция осуществляет поиск в массиве символов известной длины, пока не будет найден элемент с заданным ключом:

```

/* Последовательный поиск */
int sequential_search(char *items, int count, char key)
{
    register int t;

    for(t=0; t < count; ++t)
        if(key == items[t]) return t;
    return -1; /* ключ не найден */
}

```

Здесь `items` — указатель на массив, содержащий информацию. Функция возвращает индекс подходящего элемента, если таковой существует, либо `-1` в противном случае.

Понятно, что последовательный поиск в среднем просматривает $n/2$ элементов. В лучшем случае он проверяет только один элемент, а в худшем — n . Если информация хранится на диске, поиск может занимать продолжительное время. Но если данные не упорядочены, последовательный поиск — единственно возможный метод.

Двоичный поиск

Если данные, в которых производится поиск, отсортированы, для нахождения элемента можно применять метод, намного превосходящий предыдущий — *двоичный поиск*¹. В нем применяется метод половинного деления. Сначала проверим средний элемент. Если он больше, чем искомый ключ, проверим средний элемент первой половины, в противном случае — средний элемент второй половины. Будем повторять эту процедуру до тех пор, пока искомый элемент не будет найден либо пока не останется очередного элемента.

Например, чтобы найти число 4 в массиве

1 2 3 4 5 6 7 8 9

при двоичном поиске сначала проверяется средний элемент — число 5. Поскольку оно больше, чем 4, поиск продолжается в первой половине:

1 2 3 4 5

Средний элемент теперь равен 3. Это меньше, чем 4, поэтому первая половина отбрасывается. Поиск продолжается в части

4 5

На этот раз искомый элемент найден.

В двоичном поиске количество сравнений в худшем случае равно

$\log_2 n$

В среднем случае количество немного ниже, а в лучшем — количество сравнений равно 1.

Ниже приведена функция двоичного поиска для массивов символов. Этот поиск можно адаптировать для произвольных структур данных, изменив фрагмент сравнения.

```
/* Двоичный поиск. */
int binary_search(char *items, int count, char key)
{
    int low, high, mid;

    low = 0; high = count-1;
    while(low <= high) {
        mid = (low+high)/2;
        if(key < items[mid]) high = mid-1;
        else if(key > items[mid]) low = mid+1;
        else return mid; /* ключ найден */
    }
    return -1;
}
```

¹ Есть и другие названия: *дихотомический поиск*, *логарифмический поиск*, *поиск делением пополам*. Этот метод поиска данных состоит в том, что все множество данных делится пополам и определяется, в какой из половин находится искомое данное, после чего половина, в которой находится данное, в свою очередь делится пополам и т.д. Процесс продолжается до тех пор, пока очередное полученное множество не станет равным единственному данному, которое будет искомым, либо будет установлен факт отсутствия искомого данного в этом множестве. — *Прим. ред.*

Полный
справочник по



Глава 22

**Очереди, стеки, связанные
списки и деревья**

Как известно, программы состоят из двух частей — алгоритмов и структур данных. В хорошей программе эти составляющие эффективно дополняют друг друга. Выбор и реализация структуры данных несколько же важны, как и процедуры для обработки данных. Способ организации и доступа к информации обычно определяется природой программируемой задачи. Таким образом, для программиста важно иметь в своем распоряжении приемы, подходящие для различных ситуаций.

Степень привязки типа данных к своему машинному представлению находится в обратной зависимости от его абстракции. Другими словами, чем более абстрактными становятся типы данных, тем больше концептуальное представление о способе хранения этих данных отличается от реального, фактического способа их хранения в памяти компьютера. Простые типы, например, `char` или `int`, тесно связаны со своим машинным представлением. Например, машинное представление целочисленного значения хорошо аппроксимирует соответствующую концепцию программирования. По мере своего усложнения типы данных становятся концептуально менее похожими на свои машинные эквиваленты. Так, действительные числа с плавающей точкой более абстрактны, чем целые числа. Фактическое представление типа `float` в машине весьма приблизительно соответствует представлению среднего программиста о действительном числе. Еще более абстрактной является структура, принадлежащая к составным типам данных.

На следующем уровне абстракции сугубо физические аспекты данных отходят на второй план вследствие введения *механизма доступа* (*data engine*) к данным, то есть механизма *сохранения* и *получения* информации. По существу, физические данные связываются с механизмом доступа, который управляет работой с данными из программы. Именно механизм доступа к данным и посвящена эта глава.

Существует четыре механизма доступа:

- Очередь (*queue*)
- Стек (*stack*)¹
- Связанный список (*linked list*)²
- Двоичное дерево (*binary tree*)³

Каждый из этих методов дает возможность решать задачи определенного класса. Эти методы по существу являются механизмами, выполняющими определенные операции сохранения и получения передаваемой им информации на основе получаемых ими запросов. Они все сохраняют и получают элемент, здесь под элементом подразумевается информационная единица. В этой главе показано, как создавать такие механизмы доступа на языке С. При этом проиллюстрированы некоторые распространенные приемы программирования в С, включая динамическое выделение памяти и использование указателей.



Очереди

Очередь — это линейный список информации, работа с которой происходит по принципу “первым пришел — первым вышел” (*first-in, first-out*); этот принцип (и очередь как структура данных) иногда еще называется *FIFO*⁴. Это значит, что первый помещенный в очередь элемент будет получен из нее первым, второй помещенный

¹ Другие названия: *магазин*, *стековая память*, *магазинная память*, *память магазинного типа*, *запоминающее устройство магазинного типа*, *стековое запоминающее устройство*. — Прим. ред.

² Другие названия: *цепной список*, *список с использованием указателей*, *список со ссылками*, *список на указателях*. — Прим. ред.

³ Другие названия: *дерево двоичного поиска*. — Прим. ред.

⁴ Этот принцип имеет и другие названия: “*первым пришел — первым обслужен*”, “*в порядке поступления*”, “*первым пришел — первым вышел*”, “*обратного магазинного типа*”. — Прим. ред.

элемент будет извлечен вторым и т.д. Это единственный способ работы с очередью; произвольный доступ к отдельным элементам не разрешается.

Очереди очень часто встречаются в реальной жизни, например, около банков или ресторанов быстрого обслуживания. Чтобы представить себе работу очереди, давайте введем две функции: `qstore()` и `qretrieve()` (от “store” — “сохранять”, “retrieve” — “получать”). Функция `qstore()` помещает элемент в конец очереди, а функция `qretrieve()` удаляет элемент из начала очереди и возвращает его значение. В табл. 22.1 показано действие последовательности таких операций.

Следует иметь в виду, что операция извлечения удаляет элемент из очереди и уничтожает его, если он не хранится где-нибудь в другом месте. Поэтому после извлечения всех элементов очередь будет пуста.

В программировании очереди применяются при решении многих задач. Один из наиболее популярных видов таких задач — симуляция. Очереди также применяются в планировщиках задач операционных систем и при буферизации ввода/вывода.

Чтобы проиллюстрировать работу очереди, мы напишем простую программу планирования встреч. Эта программа позволяет сохранять информацию о некотором количестве встреч; потом по мере прохождения каждой встречи она удаляется из списка. Для упрощения описание встреч ограничено 255 символами, а количество встреч — произвольным числом 100.

При разработке этой простой программы планирования необходимо прежде всего реализовать описанные здесь функции `qstore()` и `qretrieve()`. Они будут хранить указатели на строки, содержащие описания встреч.

Таблица 22.1. Работа очереди

Действие	Содержимое очереди
<code>qstore(A)</code>	A
<code>qstore(B)</code>	A B
<code>qstore(C)</code>	A B C
<code>qretrieve()</code> возвращает A	B C
<code>qstore(D)</code>	B C D
<code>qretrieve()</code> возвращает B	C D
<code>qretrieve()</code> возвращает C	D

```
#define MAX 100

char *p[MAX];
int spos = 0;
int rpos = 0;

/* Сохранение встречи. */
void qstore(char *q)
{
    if(spos==MAX) {
        printf("Список переполнен\n");
        return;
    }
    p[spos] = q;
    spos++;
}

/* Получение встречи. */
char *qretrieve()
{
    if(rpos==spos) {
```

```

    printf("Встреч больше нет.\n");
    return '\0';
}
rpos++;
return p[rpos-1];
}

```

Обратите внимание, что этим двум функциям требуются две глобальные переменные: `spos`, в которой хранится индекс следующего свободного места в списке, и `rpos`, в которой хранится индекс следующего элемента, подлежащего выборке. С помощью этих функций можно организовать очередь данных другого типа, просто поменяв базовый тип обрабатываемого ими массива.

Функция `qstore()` помещает описание новых встреч в конец списка и проверяет, не переполнен ли список. Функция `qretrieve()` извлекает встречи из очереди, если таковые имеются. При назначении встреч увеличивается значение переменной `spos`, а по мере их прохождения увеличивается значение переменной `rpos`. По существу, `rpos` “догоняет” `spos` в очереди. На рис 22.1 показано, что может происходить в памяти при выполнении программы. Если `rpos` и `spos` равны, назначенные события отсутствуют. Даже несмотря на то, что функция `qretrieve()` не уничтожает хранящуюся в очереди информацию физически, эту информацию можно считать уничтоженной, так как повторно получить доступ к ней невозможно.

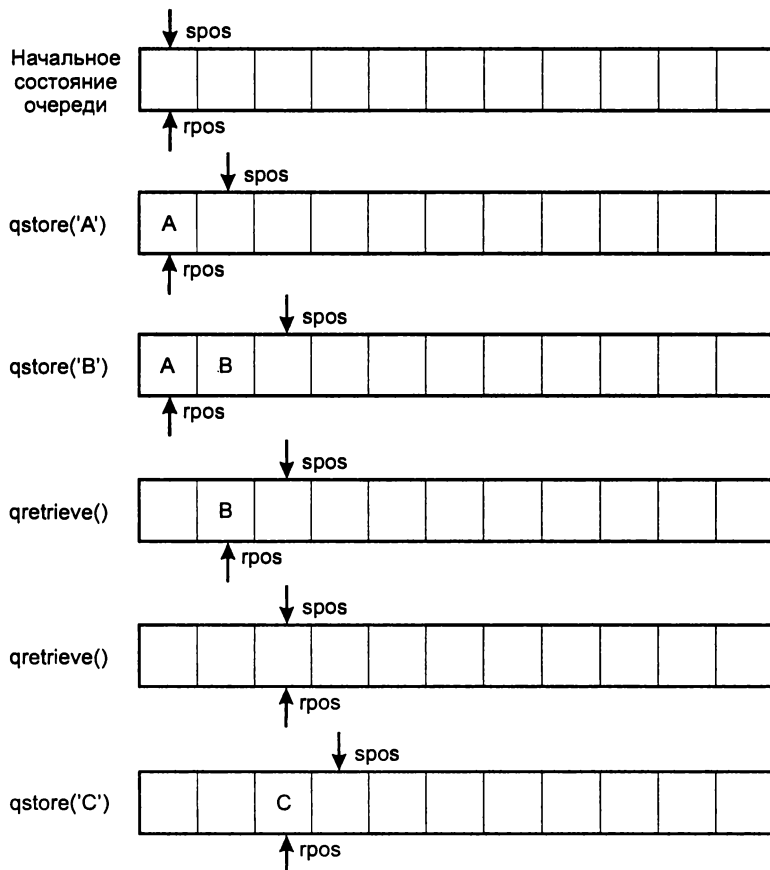


Рис. 22.1. Индекс выборки “догоняет” индекс вставки

Текст программы простого планировщика встреч целиком приведен ниже. Вы можете доработать эту программу по своему усмотрению.

```
/* Мини-планировщик событий */

#include <string.h>
#include <stdlib.h>
#include <stdio.h>
#include <ctype.h>

#define MAX 100

char *p[MAX], *qretrieve(void);
int spos = 0;
int rpos = 0;
void enter(void), qstore(char *q), review(void), delete_ap(void);

int main(void)
{
    char s[80];
    register int t;

    for(t=0; t < MAX; ++t) p[t] = NULL; /* инициализировать массив
                                           пустыми указателями */

    for(;;) {
        printf("Ввести (E), Список (L), Удалить (R), Выход (Q): ");
        gets(s);
        *s = toupper(*s);

        switch(*s) {
            case 'E':
                enter();
                break;
            case 'L':
                review();
                break;
            case 'R':
                delete_ap();
                break;
            case 'Q':
                exit(0);
        }
    }
    return 0;
}

/* Вставка в очередь новой встречи. */
void enter(void)
{
    char s[256], *p;

    do {
        printf("Введите встречу %d: ", spos+1);
        gets(s);
        if(*s==0) break; /* запись не была произведена */
        p = (char *) malloc(strlen(s)+1);
        if(!p) {
```

```

        printf("Не хватает памяти.\n");
        return;
    }
    strcpy(p, s);
    if(*s) qstore(p);
} while(*s);
}

/* Просмотр содержимого очереди. */
void review(void)
{
    register int t;

    for(t=rpos; t < spos; ++t)
        printf("%d. %s\n", t+1, p[t]);
}

/* Удаление встречи из очереди. */
void delete_ap(void)
{
    char *p;

    if((p=qretrieve())==NULL) return;
    printf("%s\n", p);
}

/* Вставка встречи. */
void qstore(char *q)
{
    if(spos==MAX) {
        printf("Список полон\n");
        return;
    }
    p[spos] = q;
    spos++;
}

/* Извлечение встречи. */
char *qretrieve(void)
{
    if(rpos==spos) {
        printf("Встреч больше нет.\n");
        return NULL;
    }
    rpos++;
    return p[rpos-1];
}

```



Циклическая очередь

При изучении предыдущего примера программы планирования встреч, вероятно, вам в голову мог прийти следующий способ ее улучшения: при достижении конца массива, в котором хранится очередь, можно не останавливать программу, а устанавливать индексы вставки (*spos*) и извлечения (*rpos*) так, чтобы они указывали на начало массива. Это позволит помещать в очередь любое количество элементов при условии их своевременного извлечения. Такая реализация очереди называется *цикличес-*

ской очереди, поскольку массив используется так, будто он представляет собой не линейный список, а кольцо.

Чтобы организовать в программе-планировщике циклическую очередь, функции `qstore()` и `qretrieve()` необходимо переписать следующим образом:

```
void qstore(char *q)
{
    /* Очередь переполняется, когда spos на единицу
       меньше rpos, либо когда spos указывает
       на конец массива, а rpos — на начало.
    */
    if(spos+1==rpos || (spos+1==MAX && !rpos)) {
        printf("Список полон\n");
        return;
    }

    p[spos] = q;
    spos++;
    if(spos==MAX) spos = 0; /* установить на начало */
}

char *qretrieve(void)
{
    if(rpos==MAX) rpos = 0; /* установить на начало */
    if(rpos==spos) {
        printf("Встреч больше нет.\n");
        return NULL;
    }
    rpos++;
    return p[rpos-1];
}
```

В данной версии программы очередь переполняется, когда индекс записи находится непосредственно перед индексом извлечения; в противном случае еще есть место для вставки события. Очередь пуста, когда `rpos` равняется `spos`.

Вероятно, чаще всего циклические очереди применяются в операционных системах для хранения информации, считываемой и записываемой в дисковые файлы или на консоль. Циклические очереди также используются в программах обработки реального времени, которые должны продолжать обрабатывать информацию, буферизируя при этом запросы на ввод/вывод. Многие текстовые процессоры используют этот прием во время переформатирования абзаца или выравнивания строки. Вводимый текст не отображается на экране до окончания процесса. Для этого прикладная программа должна проверять нажатие клавиш во время выполнения другой задачи. Если какая-либо клавиша была нажата, введенный символ быстро помещается в очередь, и процесс продолжается. После его завершения символы извлекаются из очереди.

Чтобы ощутить на практике данное применение циклических очередей, давайте рассмотрим простую программу, состоящую из двух процессов. Первый процесс в программе выводит на экран числа от 1 до 32 000. Второй процесс помещает символы в очередь по мере их ввода, не отображая их на экране, пока пользователь не нажмет <Enter>. Вводимые символы не отображаются, поскольку первому процессу дан приоритет в отношении вывода на экран. После нажатия <Enter> символы из очереди извлекаются и печатаются.

Чтобы программа работала, как описано выше, в ней необходимо использовать две функции, не определенные в стандартном языке C: `_kbhit()` и `_getch()`. Функция `_kbhit()` возвращает значение ИСТИНА, если на клавиатуре была нажата клавиша; в противном случае она возвращает ЛОЖЬ. Функция `_getch()` считывает введенный сим-

вол, но не дублирует его на экране. В стандарте языка С не предусмотрены функции для проверки состояния клавиатуры или считывания символов без отображения на экране, поскольку эти функции зависят от операционной системы. Тем не менее, в большинстве библиотек компиляторов есть функции, выполняющие данные задачи. Приведенная здесь небольшая программа компилируется с помощью компилятора Microsoft.

```
/* Пример циклической очереди в качестве буфера клавиатуры. */
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>

#define MAX 80

char buf[MAX+1];
int spos = 0;
int rpos = 0;

void qstore(char q);
char qretrieve(void);

int main(void)
{
    register char ch;
    int t;

    buf[80] = '\0';

    /* Принимать вводимые символы до нажатия <Enter>. */
    for(ch=' ', t=0; t<32000 && ch!='\r'; ++t) {
        if(_kbhit()) {
            ch = _getch();
            qstore(ch);
        }
        printf("%d ", t);
        if(ch == '\r') {
            /* Вывести на экран содержимое буфера клавиатуры
            и освободить буфер. */
            printf("\n");
            while((ch=qretrieve()) != '\0') printf("%c", ch);
            printf("\n");
        }
    }
    return 0;
}

/* Занесение символа в очередь. */
void qstore(char q)
{
    if(spos+1==rpos || (spos+1==MAX && !rpos)) {
        printf("Список полон\n");
        return;
    }
    buf[spos] = q;
    spos++;
    if(spos==MAX) spos = 0; /* установить на начало */
}

/* Получение символа из очереди. */
```

```
char qretrieve(void)
{
    if(rpos==MAX) rpos = 0; /* установить на начало */
    if(rpos==spos) return '\0';

    rpos++;
    return buf[rpos-1];
}
```



Стеки

Стек (stack) является как бы противоположностью очереди, поскольку он работает по принципу “последним пришел — первым вышел” (last-in, first-out, LIFO)¹. Чтобы наглядно представить себе стек, вспомните стопку тарелок. Первая тарелка, стоящая на столе, будет использована последней, а последняя тарелка, положенная наверх — первой. Стеки часто применяются в системном программном обеспечении, включая компиляторы и интерпретаторы.

При работе со стеками операции занесения и извлечения элемента являются основными. Данные операции традиционно называются “затолкать в стек” (push)² и “вытолкнуть из стека” (pop)³. Поэтому для реализации стека необходимо написать две функции: push(), которая “заталкивает” значение в стек, и pop(), которая “вытаскивает” значение из стека. Также необходимо выделить область памяти, которая будет использоваться в качестве стека. Для этой цели можно отвести массив или динамически выделить фрагмент памяти с помощью функций языка C, предусмотренных для динамического распределения памяти. Как и в случае очереди, функция извлечения получает из списка элемент и удаляет его, если он не хранится где-либо еще. Ниже приведена общая форма функций push() и pop(), работающих с целочисленным массивом. Стеки данных другого типа можно организовывать, изменив базовый тип данных массива.

```
int stack[MAX];
int tos=0; /* вершина стека */

/* Затолкать элемент в стек. */
void push(int i)
{
    if(tos >= MAX) {
        printf("Стек полон\n");
        return;
    }
    stack[tos] = i;
    tos++;
}

/* Получить верхний элемент стека. */
int pop(void)
{

```

¹ Иными словами, в магазинном порядке. — Прим. ред.

² А также: проталкивать (в стек), помещать на стек, класть в стек, поместить в стек, положить в стек, сохранить в стеке. — Прим. ред.

³ А также: вытаскивать данные из стека, выталкивание из стека, выталкивание данных из стека, снимать со стека, вынимать из стека, считывать из стека, вытаскивать из стека. — Прим. ред.

```

    tos--;
    if(tos < 0) {
        printf("Стек пуст\n");
        return 0;
    }
    return stack[tos];
}

```

Переменная *tos* (“top of stack” — “вершина стека”¹) содержит индекс вершины стека. При реализации данных функций необходимо учитывать случаи, когда стек заполнен или пуст. В нашем случае признаком пустого стека является равенство *tos* нулю, а признаком переполнения стека — такое увеличение *tos*, что его значение указывает куда-нибудь за пределы последней ячейки массива. Пример работы стека показан в табл. 22.2.

Прекрасный пример использования стека — калькулятор с четырьмя действиями. Большинство современных калькуляторов воспринимают стандартную запись выражений, называемую *инфиксной записью*², общая форма которой выглядит как *операнд-оператор-операнд*. Например, чтобы сложить 100 и 200, необходимо ввести 100, нажать кнопку “плюс” (“+”), затем ввести 200 и нажать кнопку “равно” (“=”). Напротив, во многих ранних калькуляторах (и некоторых из производимых сегодня) применяется *постфиксная запись*³, в которой сначала вводятся оба операнда, а затем оператор. Например, чтобы сложить 100 и 200 в постфиксной записи, необходимо ввести 100, затем 200, а потом нажать клавишу “плюс”. В этом методе операнды при вводе заталкиваются в стек. При вводе оператора операнды извлекаются (вытаскиваются) из стека, а результат помещается обратно в стек. Одно из преимуществ постфиксной формы заключается в легкости ввода длинных сложных выражений.

Следующий пример демонстрирует использование стека в программе, реализующей постфиксный калькулятор для целочисленных выражений. Для начала необходимо модифицировать функции *push()* и *pop()*, как показано ниже. Следует знать, что стек будет размещаться в динамически распределяемой памяти, а не в массиве фиксированного размера. Хотя применение динамического распределения памяти и не требуется в таком простом примере, мы увидим, как использовать динамическую память для хранения данных стека.

Таблица 22.2. Действие стека

Действие	Содержимое стека
<i>push(A)</i>	A
<i>push(B)</i>	B A
<i>push(C)</i>	C B A
<i>pop()</i> извлекает C	B A
<i>push(F)</i>	F B A
<i>pop()</i> извлекает F	B A
<i>pop()</i> извлекает B	A
<i>pop()</i> извлекает A	пусто

```

int *p; /* указатель на область свободной памяти */
int *tos; /* указатель на вершину стека */
int *bos; /* указатель на дно стека */

```

¹ Называется также *верхушкой*. — Прим. ред.

² Другие названия: *инфиксное представление*, *инфиксная нотация*. — Прим. ред.

³ Другие названия: *постфиксная нотация*, *польская инверсная запись*. — Прим. ред.


```

/* Занесение элемента в стек. */
void push(int i)
{
    if(p > bos) {
        printf("Стек полон\n");
        return;
    }
    *p = i;
    p++;
}

/* Получение верхнего элемента из стека. */
int pop(void)
{
    p--;
    if(p < tos) {
        printf("Стек пуст\n");
        return 0;
    }
    return *p;
}

```

Перед использованием этих функций необходимо выделить память из области свободной памяти с помощью функции `malloc()` и присвоить переменной `tos` адрес начала этой области, а переменной `bos` — адрес ее конца.

Текст программы постфиксного калькулятора целиком приведен ниже.

```

/* Простой калькулятор с четырьмя действиями. */

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

#define MAX 100

int *p; /* указатель на область свободной памяти */
int *tos; /* указатель на вершину стека */
int *bos; /* указатель на дно стека */

void push(int i);
int pop(void);

int main(void)
{
    int a, b;
    char s[80];

    p = (int *) malloc(MAX*sizeof(int)); /* получить память для стека */
    if(!p) {
        printf("Ошибка при выделении памяти!\n");
        exit(1);
    }
    tos = p;
    bos = p + MAX-1;

    printf("Калькулятор с четырьмя действиями\n");
    printf("Нажмите 'q' для выхода\n");

    do {
        printf(": ");

```

```

    gets(s);
    switch(*s) {
        case '+':
            a = pop();
            b = pop();
            printf("%d\n", a+b);
            push(a+b);
            break;
        case '-':
            a = pop();
            b = pop();
            printf("%d\n", b-a);
            push(b-a);
            break;
        case '*':
            a = pop();
            b = pop();
            printf("%d\n", b*a);
            push(b*a);
            break;
        case '/':
            a = pop();
            b = pop();
            if(a==0) {
                printf("Деление на 0.\n");
                break;
            }
            printf("%d\n", b/a);
            push(b/a);
            break;
        case '.': /* показать содержимое вершины стека */
            a = pop();
            push(a);
            printf("Текущее значение на вершине стека: %d\n", a);
            break;
        default:
            push(atoi(s));
    }
} while(*s != 'q');

return 0;
}

/* Занесение элемента в стек. */
void push(int i)
{
    if(p > bos) {
        printf("Стек полон\n");
        return;
    }
    *p = i;
    p++;
}

/* Получение верхнего элемента из стека. */
int pop(void)
{
    p--;

```

```

if(p < tos) {
    printf("Стек пуст\n");
    return 0;
}
return *p;
}

```

■ Связанные списки

Очереди и стеки имеют две характерные особенности: обе структуры данных имеют строгие правила доступа к хранящимся в них данным, причем в результате выполнения операций извлечения данные, в сущности, уничтожаются. Другими словами, доступ к элементу в стеке или очереди приводит к его удалению, и если этот элемент не сохранить где-либо в другом месте, он теряется. Кроме того, и в стеке, и в очереди используется один последовательный участок памяти. В отличие от стека или очереди, *связанный список* допускает гибкие способы доступа, поскольку каждый фрагмент информации имеет ссылку на следующий элемент данных в цепочке. Кроме того, операция извлечения не приводит к удалению из списка и уничтожению элемента данных. В принципе, для этой цели необходимо ввести дополнительную специальную операцию удаления.

Связанные списки могут быть односвязными и двусвязными¹. Односвязный список содержит ссылку на следующий элемент данных. Двусвязный список содержит ссылки как на последующий, так и на предыдущий элементы списка. Выбор типа применяемого списка зависит от конкретной задачи.

■ Односвязные списки

В односвязном списке каждый элемент информации содержит ссылку на следующий элемент списка. Каждый элемент данных обычно представляет собой структуру, которая состоит из информационных полей и указателя связи. Концептуально односвязный список выглядит так, как показано на рис. 22.2.

Существует два основных способа построения односвязного списка. Первый способ — помещать новые элементы в конец списка². Второй — вставлять элементы в определенные позиции списка, например, в порядке возрастания. От способа построения списка зависит алгоритм функции добавления элемента. Давайте начнем с более простого способа создания списка путем помещения элементов в конец.

Как правило, элементы связанного списка являются структурами, так как, помимо данных, они содержат ссылку на следующий элемент. Поэтому необходимо определить структуру, которая будет использоваться в последующих примерах. Поскольку списки рассылки обычно хранятся в связанных списках, хорошим выбором будет структура, описывающая почтовый адрес. Ее описание показано ниже:

```

struct address {
    char name[40];
    char street[40];
    char city[20];
}

```

¹ Связанные списки часто называются *связными*. Односвязные списки называются еще *односвязными линейными списками*, *однонаправленными списками*, а также *однонаправленными цепочками*. Двусвязные списки иногда называются также *дважды связанными*; кроме того, их называют *двусвязными линейными списками*, а также *двунаправленными цепочками*. — Прим. ред.

² Не забывайте, что у односвязного списка, как и у веревки, два конца: начало и конец. — Прим. ред.

```

char state[3];
char zip[11];
struct address *next; /* ссылка на следующий адрес */
} info;

```

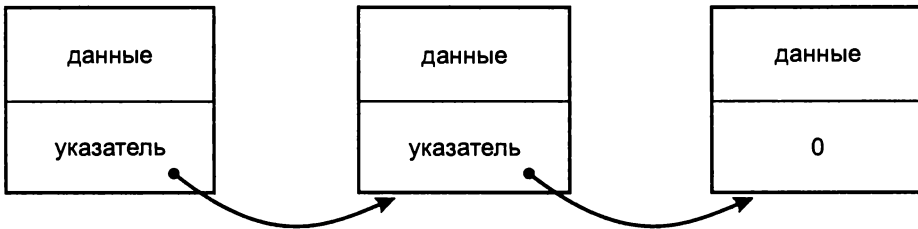


Рис. 22.2. Односвязный список

Приведенная ниже функция `slstore()` создает односвязный список путем помещения каждого очередного элемента в конец списка. В качестве параметров ей передаются указатель на структуру типа `address`, содержащую новую запись, и указатель на последний элемент списка. Если список пуст, указатель на последний элемент должен быть равен нулю.

```

void slstore(struct address *i,
             struct address **last)
{
    if(!*last) *last = i; /* первый элемент в списке */
    else (*last)->next = i;
    i->next = NULL;
    *last = i;
}

```

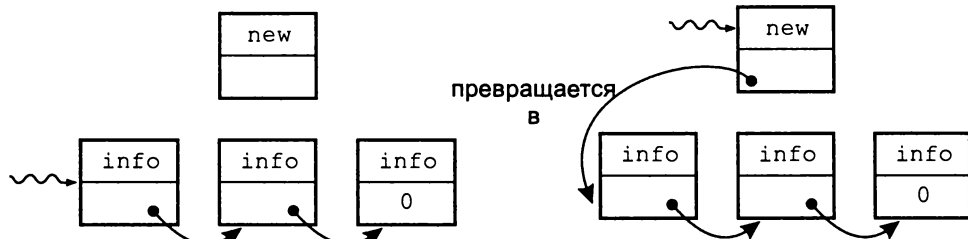
Несмотря на то, что созданный с помощью функции `slstore()` список можно отсортировать отдельной операцией уже после его создания, легче сразу создавать упорядоченный список, вставляя каждый новый элемент в нужное место в последовательности. Кроме того, если список уже отсортирован, имеет смысл поддерживать его упорядоченность, вставляя новые элементы в соответствующие позиции. Для вставки элемента таким способом требуется последовательно просматривать список до тех пор, пока не будет найдено место нового элемента, затем вставить в найденную позицию новую запись и переустановить ссылки.

При вставке элемента в односвязный список может возникнуть одна из трех ситуаций: элемент становится первым, элемент вставляется между двумя другими, элемент становится последним. На рис. 22.3 показана схема изменения ссылок в каждом случае.

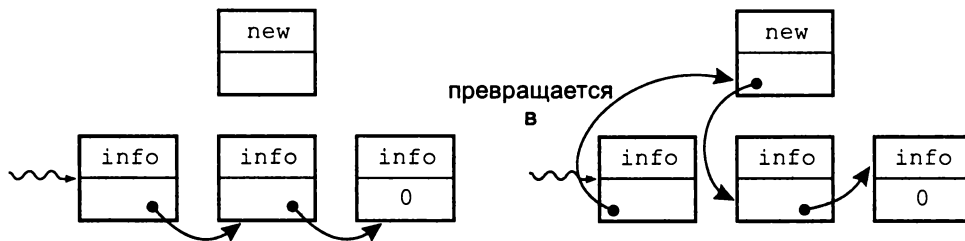
Следует помнить, что при вставке элемента в начало (первую позицию) списка необходимо также изменить адрес входа в список где-то в другом месте программы. Чтобы избежать этих сложностей, можно в качестве первого элемента списка хранить служебный *сторожевой* элемент¹. В случае упорядоченного списка необходимо выбрать некоторое специальное значение, которое всегда будет первым в списке, чтобы начальный элемент оставался неизменным. Недостатком данного метода является довольно большой расход памяти на хранение сторожевого элемента, но обычно это не столь важно.

¹ Часто называется еще *сигнальной меткой*. — Прим. ред.

Вставка в начало списка



Вставка в середину списка



Вставка в конец списка

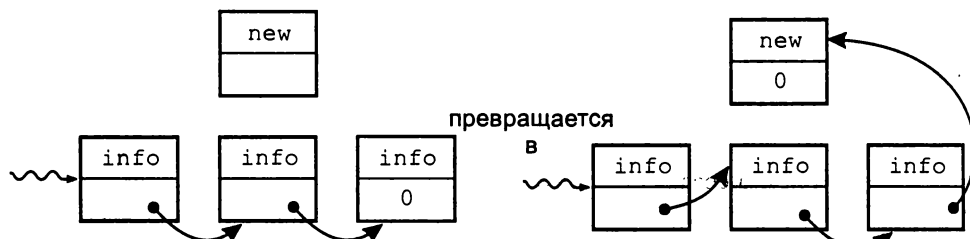


Рис. 22.3. Вставка нового элемента *new* в односвязный список (в котором *info* — поле данных)

Следующая функция, `sls_store()`, вставляет структуры типа `address` в список рассылки, упорядочивая его по возрастанию значений в поле `name`. Функция принимает указатели на указатели на первый и последний элементы списка, а также указатель на вставляемый элемент. Поскольку первый или последний элементы списка могут измениться, функция `sls_store()` при необходимости автоматически обновляет указатели на начало и конец списка. При первом вызове данной функции указатели `first` и `last` должны быть равны нулю.

```
/* Вставка в упорядоченный односвязный список. */
void sls_store(struct address *i, /* новый элемент */
               struct address **start, /* начало списка */
               struct address **last) /* конец списка */
{
    struct address *old, *p;

    p = *start;

    if(!*last) { /* первый элемент в списке */
        i->next = NULL;
```

```

        *last = i;
        *start = i;
        return;
    }

    old = NULL;
    while(p) {
        if(strcmp(p->name, i->name)<0) {
            old = p;
            p = p->next;
        }
        else {
            if(old) { /* вставка в середину */
                old->next = i;
                i->next = p;
                return;
            }
            i->next = p; /* вставка в начало */
            *start = i;
            return;
        }
    }
    (*last)->next = i; /* вставка в конец */
    i->next = NULL;
    *last = i;
}

```

Последовательный перебор элементов связанного списка осуществляется очень просто: начать с начала и следовать указателям. Обычно фрагмент кода перебора настолько мал, что его вставляют в другую процедуру — например, функцию поиска, удаления или отображения содержимого. Так, приведенная ниже функция выводит на экран все имена из списка рассылки:

```

void display(struct address *start)
{
    while(start) {
        printf("%s\n", start->name);
        start = start->next;
    }
}

```

При вызове функции `display()` параметр `start` должен быть указателем на первую структуру в списке. После этого функция переходит к следующему элементу, на который указывает указатель в поле `next`. Процесс прекращается, когда `next` равно нулю.

Для получения элемента из списка нужно просто пройти по цепочке ссылок. Ниже приведен пример функции поиска по содержимому поля `name`:

```

struct address *search(struct address *start, char *n)
{
    while(start) {
        if(!strcmp(n, start->name)) return start;
        start = start->next;
    }
    return NULL; /* подходящий элемент не найден */
}

```

Поскольку функция `search()` возвращает указатель на элемент списка, содержащий искомое имя, возвращаемый тип описан как указатель на структуру `address`. При отсутствии в списке подходящих элементов возвращается ноль (`NULL`).

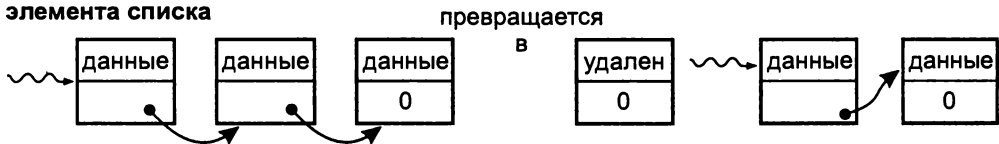
Удаление элемента из односвязного списка выполняется просто. Так же, как и при вставке, возможны три случая: удаление первого элемента, удаление элемента в середине, удаление последнего элемента. На рис. 22.4 показаны все три операции.

Ниже приведена функция, удаляющая заданный элемент из списка структур address.

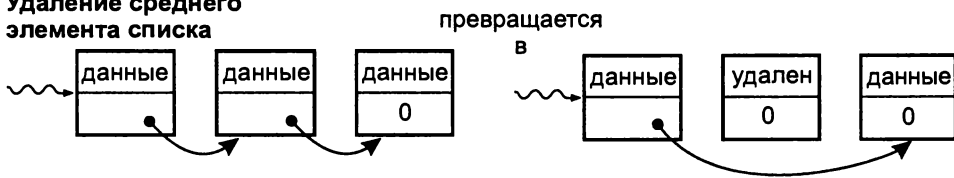
```
void sdelete(
    struct address *p, /* предыдущий элемент */
    struct address *i, /* удаляемый элемент */
    struct address **start, /* начало списка */
    struct address **last) /* конец списка */
{
    if(p) p->next = i->next;
    else *start = i->next;

    if(i==*last && p) *last = p;
}
```

Удаление первого элемента списка



Удаление среднего элемента списка



Удаление последнего элемента списка

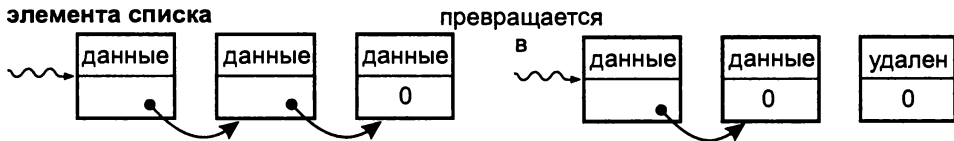


Рис. 22.4. Удаление элемента из односвязного списка

Функции `sdelete()` необходимо передавать указатели на удаляемый элемент, предшествующий удаляемому, а также на первый и последний элементы. При удалении первого элемента указатель на предшествующий элемент должен быть равен нулю (NULL). Данная функция автоматически обновляет указатели `start` и `last`, если один из них ссылается на удаляемый элемент.

У односвязных списков есть один большой недостаток: односвязный список невозможно прочитать в обратном направлении. По этой причине обычно применяются двусвязные списки.



Двусвязные списки

Двусвязный список состоит из элементов данных, каждый из которых содержит ссылки как на следующий, так и на предыдущий элементы. На рис. 22.5 показана организация ссылок в двусвязном списке.

Наличие двух ссылок вместо одной предоставляет несколько преимуществ. Вероятно, наиболее важное из них состоит в том, что перемещение по списку возможно в обоих направлениях. Это упрощает работу со списком, в частности, вставку и удаление. Помимо этого, пользователь может просматривать список в любом направлении. Еще одно преимущество имеет значение только при некоторых сбоях. Поскольку весь список можно пройти не только по прямым, но и по обратным ссылкам, то в случае, если какая-то из ссылок станет неверной, целостность списка можно восстановить по другой ссылке.

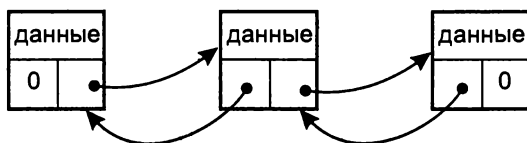


Рис. 22.5. Двусвязный список

При вставке нового элемента в двусвязный список могут быть три случая: элемент вставляется в начало, в середину и в конец списка. Эти операции показаны на рис. 22.6.

Построение двусвязного списка выполняется аналогично построению односвязного за исключением того, что необходимо установить две ссылки. Поэтому в структуре данных должны быть описаны два указателя связи. Возвращаясь к примеру списка рассылки, для двусвязного списка структуру `address` можно модифицировать следующим образом:

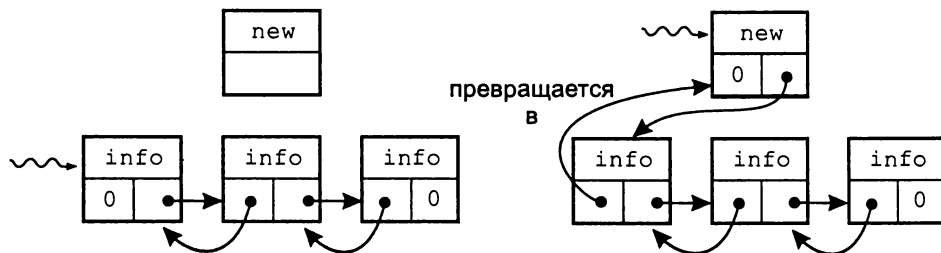
```
struct address {
    char name[40];
    char street[40];
    char city[20];
    char state[3];
    char zip[11];
    struct address *next;
    struct address *prior;
} info;
```

Следующая функция, `dlstore()`, создает двусвязный список, используя структуру `address` в качестве базового типа данных:

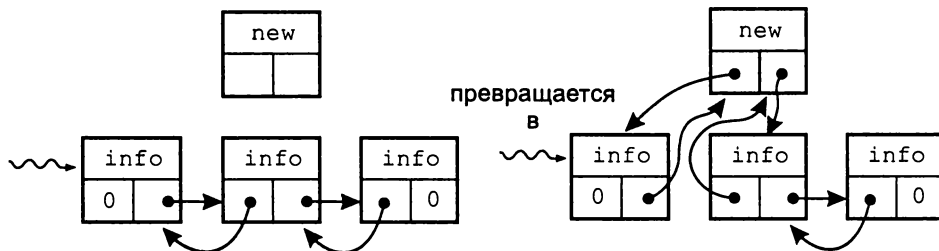
```
void dlstore(struct address *i, struct address **last)
{
    if(*last) *last = i; /* вставка первого элемента */
    else (*last)->next = i;
    i->next = NULL;
    i->prior = *last;
    *last = i;
}
```

Функция `dlstore()` помещает новые записи в конец списка. В качестве параметров ей необходимо передавать указатель на сохраняемые данные, а также указатель на конец списка, который при первом вызове должен быть равен нулю (`NULL`).

Вставка элемента в начало списка



Вставка элемента в середину списка



Вставка элемента в конец списка

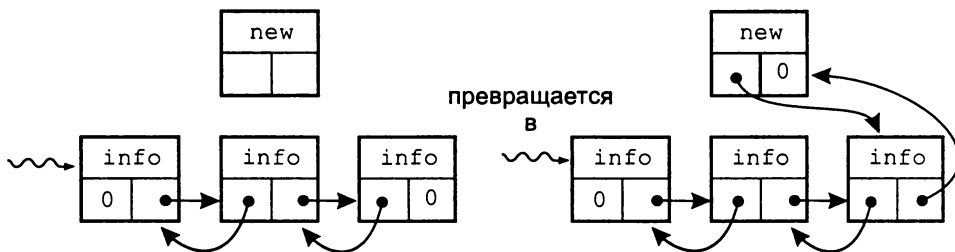


Рис. 22.6. Операции с двусвязным списком. (Здесь new — вставляемый элемент, а info — поле данных)

Подобно односвязным, двусвязные списки можно создавать с помощью функции, которая будет помещать элементы в определенные позиции, а не только в конец списка. Показанная ниже функция `dls_store()` создает список, упорядочивая его по возрастанию имен:

```
/* Создание упорядоченного двусвязного списка. */
void dls_store(
    struct address *i, /* новый элемент */
    struct address **start, /* первый элемент в списке */
    struct address **last /* последний элемент в списке */
)
{
    struct address *old, *p;

    if(*last==NULL) { /* первый элемент в списке */
        i->next = NULL;
        i->prior = NULL;
        *last = i;
        *start = i;
    }
```

```

    return;
}

p = *start; /* начать с начала списка */

old = NULL;
while(p) {
    if(strcmp(p->name, i->name)<0){
        old = p;
        p = p->next;
    }
    else {
        if(p->prior) {
            p->prior->next = i;
            i->next = p;
            i->prior = p->prior;
            p->prior = i;
            return;
        }
        i->next = p; /* новый первый элемент */
        i->prior = NULL;
        p->prior = i;
        *start = i;
        return;
    }
}
old->next = i; /* вставка в конец */
i->next = NULL;
i->prior = old;
*last = i;
}

```

Поскольку первый и последний элементы списка могут меняться, функция `dls_store()` автоматически обновляет указатели на начало и конец списка посредством параметров `start` и `last`. При вызове функции необходимо передавать указатель на сохраняемые данные и указатели на первый и последний элементы списка. В первый раз параметры `start` и `last` должны быть равны нулю (`NULL`).

Как и в односвязных списках, для получения элемента данных двусвязного списка необходимо переходить по ссылкам до тех пор, пока не будет найден искомый элемент.

При удалении элемента двусвязного списка могут возникнуть три случая: удаление первого элемента, удаление элемента из середины и удаление последнего элемента. На рис. 22.7 показано, как при этом изменяются ссылки. Показанная ниже функция `dldelete()` удаляет элемент двусвязного списка:

```

void dldelete(
    struct address *i, /* удаляемый элемент */
    struct address **start, /* первый элемент */
    struct address **last) /* последний элемент */
{
    if(i->prior) i->prior->next = i->next;
    else { /* удаление первого элемента */
        *start = i->next;
        if(start) start->prior = NULL;
    }

    if(i->next) i->next->prior = i->prior;
    else /* удаление последнего элемента */
        *last = i->prior;
}

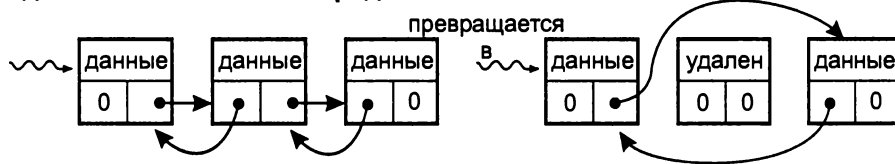
```

Поскольку первый или последний элементы списка могут быть удалены, функция `ddelete()` автоматически обновляет указатели на начало и конец списка посредством параметров `start` и `last`. При вызове ей передаются указатель на удаляемый элемент и указатели на начало и конец списка.

Удаление первого элемента списка



Удаление элемента из середины списка



Удаление последнего элемента списка

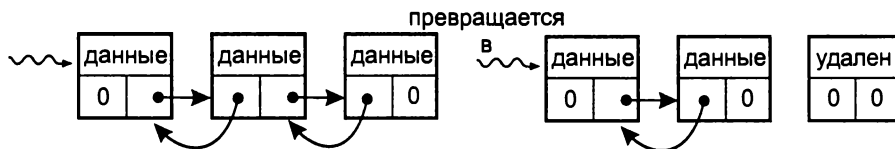


Рис. 22.7. Удаление элемента двусвязного списка

Пример списка рассылки

Чтобы завершить обсуждение двусвязных списков, в данном разделе представлена простая, но законченная программа для работы со списком рассылки. Во время работы весь список хранится в оперативной памяти. Тем не менее, его можно сохранять в файле и загружать для дальнейшей работы.

```
/* Простая программа для обработки списка рассылки,
   иллюстрирующая работу с двусвязными списками.
*/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

struct address {
    char name[30];
    char street[40];
    char city[20];
    char state[3];
    char zip[11];
    struct address *next; /* указатель на следующую запись */
    struct address *prior; /* указатель на предыдущую запись */
};

struct address *start; /* указатель на первую запись списка */
```

```

struct address *last; /* указатель на последнюю запись */
struct address *find(char *);

void enter(void), search(void), save(void);
void load(void), list(void);
void mdelete(struct address **, struct address **);
void dls_store(struct address *i, struct address **start,
               struct address **last);
void inputs(char *, char *, int), display(struct address *);
int menu_select(void);

int main(void)
{
    start = last = NULL; /* инициализация указателей на начало и конец */

    for(;;) {
        switch(menu_select()) {
            case 1: enter(); /* ввод адреса */
                    break;
            case 2: mdelete(&start, &last); /* удаление адреса */
                    break;
            case 3: list(); /* отображение списка */
                    break;
            case 4: search(); /* поиск адреса */
                    break;
            case 5: save(); /* запись списка в файл */
                    break;
            case 6: load(); /* считывание с диска */
                    break;
            case 7: exit(0);
        }
    }
    return 0;
}

/* Выбор действия пользователя. */
int menu_select(void)
{
    char s[80];
    int c;

    printf("1. Ввод имени\n");
    printf("2. Удаление имени\n");
    printf("3. Отображение содержимого списка\n");
    printf("4. Поиск\n");
    printf("5. Сохранить в файл\n");
    printf("6. Загрузить из файла\n");
    printf("7. Выход\n");
    do {
        printf("\nВаш выбор: ");
        gets(s);
        c = atoi(s);
    } while(c<0 || c>7);
    return c;
}

/* Ввод имен и адресов. */
void enter(void)
{

```

```

struct address *info;

for(;;) {
    info = (struct address *)malloc(sizeof(struct address));
    if(!info) {
        printf("\nНет свободной памяти");
        return;
    }

    inputs("Введите имя: ", info->name, 30);
    if(!info->name[0]) break; /* завершить ввод */
    inputs("Введите улицу: ", info->street, 40);
    inputs("Введите город: ", info->city, 20);
    inputs("Введите штат: ", info->state, 3);
    inputs("Введите почтовый индекс: ", info->zip, 10);

    dls_store(info, &start, &last);
} /* Цикл ввода */

/* Следующая функция вводит с клавиатуры строку
   длиной не больше count и предотвращает переполнение
   строки. Кроме того, она выводит на экран подсказку. */
void inputs(char *prompt, char *s, int count)
{
    char p[255];

    do {
        printf(prompt);
        fgets(p, 254, stdin);
        if(strlen(p) > count) printf("\nСлишком длинная строка\n");
    } while(strlen(p) > count);

    p[strlen(p)-1] = 0; /* удалить символ перевода строки */
    strcpy(s, p);
}

/* Создание упорядоченного двусвязного списка. */
void dls_store(
    struct address *i, /* новый элемент */
    struct address **start, /* первый элемент списка */
    struct address **last /* последний элемент списка */
)
{
    struct address *old, *p;

    if(*last==NULL) { /* первый элемент списка */
        i->next = NULL;
        i->prior = NULL;
        *last = i;
        *start = i;
        return;
    }
    p = *start; /* начать с начала списка */

    old = NULL;
    while(p) {
        if(strcmp(p->name, i->name)<0) {
            old = p;

```

```

        p = p->next;
    }
    else {
        if(p->prior) {
            p->prior->next = i;
            i->next = p;
            i->prior = p->prior;
            p->prior = i;
            return;
        }
        i->next = p; /* новый первый элемент */
        i->prior = NULL;
        p->prior = i;
        *start = i;
        return;
    }
}
old->next = i; /* вставка в конец */
i->next = NULL;
i->prior = old;
*last = i;
}

/* Удаление элемента из списка. */
void mdelete(struct address **start, struct address **last)
{
    struct address *info;
    char s[80];

    inputs("Введите имя: ", s, 30);
    info = find(s);
    if(info) {
        if(*start==info) {
            *start=info->next;
            if(*start) (*start)->prior = NULL;
            else *last = NULL;
        }
        else {
            info->prior->next = info->next;
            if(info!=*last)
                info->next->prior = info->prior;
            else
                *last = info->prior;
        }
        free(info); /* освободить память */
    }
}

/* Поиск адреса. */
struct address *find( char *name)
{
    struct address *info;

    info = start;
    while(info) {
        if(!strcmp(name, info->name)) return info;
        info = info->next; /* перейти к следующему адресу */
    }
    printf("Имя не найдено.\n");
}

```

```

    return NULL; /* нет подходящего элемента */
}

/* Отобразить на экране весь список. */
void list(void)
{
    struct address *info;

    info = start;
    while(info) {
        display(info);
        info = info->next; /* перейти к следующему адресу */
    }
    printf("\n\n");
}

/* Данная функция выполняет собственно вывод на экран
   всех полей записи, содержащей адрес. */
void display(struct address *info)
{
    printf("%s\n", info->name);
    printf("%s\n", info->street);
    printf("%s\n", info->city);
    printf("%s\n", info->state);
    printf("%s\n", info->zip);
    printf("\n\n");
}

/* Поиск имени в списке. */
void search(void)
{
    char name[40];
    struct address *info;

    printf("Введите имя: ");
    gets(name);
    info = find(name);
    if(!info) printf("Не найдено\n");
    else display(info);
}

/* Сохранить список в дисковом файле. */
void save(void)
{
    struct address *info;

    FILE *fp;

    fp = fopen("mlist", "wb");
    if(!fp) {
        printf("Невозможно открыть файл.\n");
        exit(1);
    }
    printf("\nСохранение в файле\n");

    info = start;
    while(info) {
        fwrite(info, sizeof(struct address), 1, fp);
        info = info->next; /* перейти к следующему адресу */
    }
}

```

```

    }
    fclose(fp);
}

/* Загрузка адресов из файла. */
void load()
{
    struct address *info;
    FILE *fp;

    fp = fopen("mlist", "rb");
    if(!fp) {
        printf("Невозможно открыть файл.\n");
        exit(1);
    }

    /* освободить память, если в памяти уже есть список */
    while(start) {
        info = start->next;
        free(info);
        start = info;
    }

    /* сбросить указатели на начало и конец */
    start = last = NULL;

    printf("\nЗагрузка из файла\n");
    while(!feof(fp)) {
        info = (struct address *) malloc(sizeof(struct address));
        if(!info) {
            printf("Нет свободной памяти");
            return;
        }
        if(1 != fread(info, sizeof(struct address), 1, fp)) break;
        dls_store(info, &start, &last);
    }
    fclose(fp);
}

```



Двоичные деревья

Напоследок мы рассмотрим структуру данных, которая называется *двоичное дерево* (binary tree). Несмотря на то, что бывает много различных типов деревьев, двоичные деревья играют особую роль, так как в отсортированном состоянии позволяют очень быстро выполнять вставку, удаление и поиск. Каждый элемент двоичного дерева состоит из информационной части и указателей на левый и правый элементы. На рис. 22.8 показано небольшое двоичное дерево.

При обсуждении деревьев применяется специальная терминология. Программисты не являются специалистами в области филологии, и поэтому терминология, применяемая в теории графов (а ведь деревья представляют собой частный случай графов!), является классическим примером неправильного употребления слов. Первый элемент дерева называется *корнем* (root). Каждый элемент данных называется *вершиной дерева* (node), а любой фрагмент дерева называется *поддеревом* (subtree). Вершина, к которой не присоединены поддеревья, называется *заключительным узлом* (terminal node) или *листом* (leaf). *Высота* (height) дерева равняется максимальному количеству уровней от

корня до листа. При работе с деревьями можно допустить, что в памяти они существуют в том же виде, что и на бумаге. Но помните, что дерево — всего лишь способ логической организации данных в памяти, а память линейна.

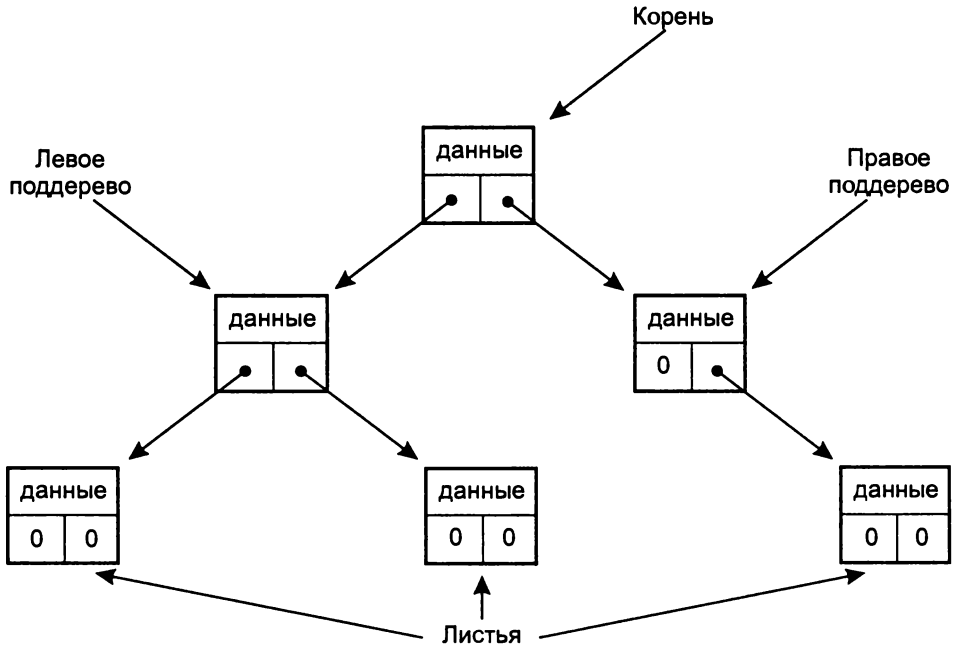
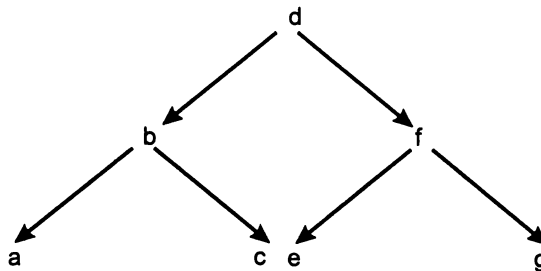


Рис. 22.8. Пример двоичного дерева, высота которого равна 3

В некотором смысле двоичное дерево является особым видом связанного списка. Элементы можно вставлять, удалять и извлекать в любом порядке. Кроме того, операция извлечения не является разрушающей. Несмотря на то, что деревья легко представить в воображении, в теории программирования с ними связан ряд сложных задач. В данном разделе деревья затрагиваются лишь поверхностно.

Большинство функций, работающих с деревьями, рекурсивны, поскольку дерево по своей сути является рекурсивной структурой данных. Другими словами, каждое поддерево, в свою очередь, является деревом. Поэтому разрабатываемые здесь функции будут рекурсивными. Существуют и не рекурсивные версии этих функций, но их код понять намного сложнее.

Способ упорядочивания дерева зависит от того, как к нему впоследствии будет осуществляться доступ. Процесс поочередного доступа к каждой вершине дерева называется *обходом (вершин) дерева* (tree traversal). Рассмотрим следующее дерево:



Существует три порядка обхода дерева: *обход симметричным способом*, или *симметричный обход* (inorder), *обход в прямом порядке*, *прямой обход*, *упорядоченный обход*, *обход сверху*, или *обход в ширину* (preorder) и *обход в обратном порядке*, *обход в глубину*, *обратный обход*, *обход снизу* (postorder). При симметричном обходе обрабатывается сначала левое поддерево, затем корень, а затем правое поддерево. При прямом обходе обрабатывается сначала корень, затем левое поддерево, а потом правое. При обходе снизу сначала обрабатывается левое поддерево, затем правое и, наконец корень. Последовательность доступа при каждом методе обхода показана ниже:

Симметричный обход	a b c d e f g
Прямой обход	d b a c f e g
Обход снизу	a c b e g f d

Несмотря на то, что дерево не должно быть обязательно упорядоченным, в большинстве задач используются именно такие деревья. Конечно, структура упорядоченного дерева зависит от способа его обхода. В оставшейся части данной главы предполагается симметричный обход. Поэтому упорядоченным двоичным деревом будет считаться такое дерево, в котором левое поддерево содержит вершины, меньшие или равные корню, а правое содержит вершины, большие корня.

Приведенная ниже функция `stree()` создает упорядоченное двоичное дерево:

```
struct tree {
    char info;
    struct tree *left;
    struct tree *right;
};

struct tree *stree(
    struct tree *root,
    struct tree *r,
    char info)
{
    if(!r) {
        r = (struct tree *) malloc(sizeof(struct tree));
        if(!r) {
            printf("Не хватает памяти\n");
            exit(0);
        }
        r->left = NULL;
        r->right = NULL;
        r->info = info;
        if(!root) return r; /* первый вход */
        if(info < root->info) root->left = r;
        else root->right = r;
        return r;
    }
    if(info < r->info)
        stree(r, r->left, info);
    else
        stree(r, r->right, info);

    return root;
}
```

Приведенный выше алгоритм просто следует по ссылкам дерева, переходя к левой или правой ветви очередной вершины на основании содержимого поля `info` до достижения места вставки нового элемента. Чтобы использовать эту функцию, необходимо иметь глобальную переменную-указатель на корень дерева. Этот указатель зна-

начально должен иметь значение нуль (NULL). При первом вызове функция `stree()` возвращает указатель на корень дерева, который нужно присвоить глобальной переменной. При последующих вызовах функция продолжает возвращать указатель на корень. Допустим, что глобальная переменная, содержащая корень дерева, называется `rt`. Тогда функция `stree()` вызывается следующим образом:

```
/* вызов функции stree() */
rt = stree(rt, rt, info);
```

Функция `stree()` использует рекурсивный алгоритм, как и большинство процедур работы с деревьями. Точно такая же функция, основанная на итеративных методах, была бы в несколько раз длиннее. Функцию `stree()` необходимо вызывать со следующими параметрами (слева направо): указатель на корень всего дерева, указатель на корень следующего поддерева, в котором осуществляется поиск, и сохраняемые данные. При первом вызове оба первых параметра указывают на корень всего дерева. Для простоты в вершинах дерева хранятся одиночные символы. Тем не менее, вместо них можно использовать любой тип данных.

Чтобы обойти созданное функцией `stree()` дерево в симметричном порядке и распечатать поле `info` в каждой вершине, можно применить приведенную ниже функцию `inorder()`:

```
void inorder(struct tree *root)
{
    if(!root) return;

    inorder(root->left);
    if(root->info) printf("%c ", root->info);
    inorder(root->right);
}
```

Данная рекурсивная функция завершает работу тогда, когда находит заключительный узел (нулевой указатель).

В следующем листинге показаны функции, выполняющие обход дерева в ширину и в глубину.

```
void preorder(struct tree *root)
{
    if(!root) return;

    if(root->info) printf("%c ", root->info);
    preorder(root->left);
    preorder(root->right);
}

void postorder(struct tree *root)
{
    if(!root) return;

    postorder(root->left);
    postorder(root->right);
    if(root->info) printf("%c ", root->info);
}
```

Теперь давайте рассмотрим короткую, но интересную программу, которая строит упорядоченное двоичное дерево, а затем, обходя его симметричным образом, отображает его на экране боком. Для отображения дерева требуется лишь слегка модифицировать функцию `inorder()`. Поскольку на экране дерево распечатывается боком, для корректного отображения правое поддерево необходимо печатать прежде левого.

(Технически это противоположность симметричного обхода.) Новая функция называется `print_tree()`, а ее код показан ниже:

```
void print_tree(struct tree *r, int l)
{
    int i;

    if(r == NULL) return;

    print_tree(r->right, l+1);
    for(i=0; i<l; ++i) printf(" ");
    printf("%c\n", r->info);
    print_tree(r->left, l+1);
}
```

Далее следует текст всей программы печати дерева. Попробуйте вводить различные деревья, чтобы увидеть, как они строятся.

```
/* Эта программа выводит на экран двоичное дерево. */

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

struct tree {
    char info;
    struct tree *left;
    struct tree *right;
};

struct tree *root; /* начальная вершина дерева */
struct tree *stree(struct tree *root,
                   struct tree *r, char info);
void print_tree(struct tree *root, int l);

int main(void)
{
    char s[80];

    root = NULL; /* инициализация корня дерева */

    do {
        printf("Введите букву: ");
        gets(s);
        root = stree(root, root, *s);
    } while(*s);

    print_tree(root, 0);

    return 0;
}

struct tree *stree(
    struct tree *root,
    struct tree *r,
    char info)
{
    if(!r) {
        r = (struct tree *) malloc(sizeof(struct tree));
        if(!r) {
```

```

        printf("Не хватает памяти\n");
        exit(0);
    }
    r->left = NULL;
    r->right = NULL;
    r->info = info;
    if(!root) return r; /* первый вход */
    if(info < root->info) root->left = r;
    else root->right = r;
    return r;
}

if(info < r->info)
    stree(r, r->left, info);
else
    stree(r, r->right, info);

return root;
}

void print_tree(struct tree *r, int l)
{
    int i;

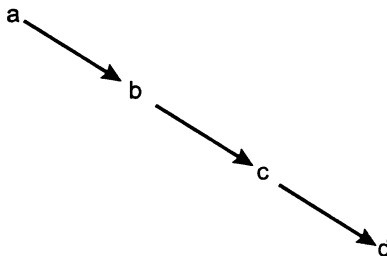
    if(!r) return;

    print_tree(r->right, l+1);
    for(i=0; i<l; ++i) printf(" ");
    printf("%c\n", r->info);
    print_tree(r->left, l+1);
}

```

По существу, данная программа сортирует вводимую информацию. Метод сортировки является одной из разновидностей сортировки методом вставок, которая была рассмотрена в предыдущей главе. В среднем случае производительность может быть вполне хорошей.

Если вы запускали программу печати дерева, вы, вероятно, заметили, что некоторые деревья являются *сбалансированными* (balanced), т.е. каждое поддерево имеет примерно такую же высоту, как и остальные, а некоторые деревья очень далеки от этого состояния. Например, дерево $a \Rightarrow b \Rightarrow c \Rightarrow d$ выглядит следующим образом:



В этом дереве нет левых поддеревьев. Такое дерево называется *вырожденным*, поскольку фактически оно выродилось в линейный список. В общем случае, если при построении дерева вводимые данные являются случайными, то получаемое дерево оказывается близким к сбалансированному. Если же информация предварительно отсортирована, создается вырожденное дерево. (Поэтому иногда при каждой вставке де-

рево корректируют так, чтобы оно было сбалансированным, но этот процесс довольно сложен и выходит за рамки данной главы.)

В двоичных деревьях легко реализовываются функции поиска. Приведенная ниже функция возвращает указатель на вершину дерева, в которой информация совпадает с ключом поиска, либо нуль (NULL), если такой вершины нет.

```
struct tree *search_tree(struct tree *root, char key)
{
    if(!root) return root; /* пустое дерево */
    while(root->info != key) {
        if(key<root->info) root = root->left;
        else root = root->right;
        if(root == NULL) break;
    }
    return root;
}
```

К сожалению, удалить вершину дерева не так просто, как отыскать. Удаляемая вершина может быть либо корнем, либо левой, либо правой вершиной. Помимо того, к вершине могут быть присоединены поддеревья (количество присоединенных поддеревьев может равняться 0, 1 или 2). Процесс переустановки указателей подсказывает рекурсивный алгоритм, приведенный ниже:

```
struct tree *dtree(struct tree *root, char key)
{
    struct tree *p,*p2;

    if(!root) return root; /* вершина не найдена */

    if(root->info == key) { /* удаление корня */
        /* это означает пустое дерево */
        if(root->left == root->right){
            free(root);
            return NULL;
        }
        /* или если одно из поддеревьев пустое */
        else if(root->left == NULL) {
            p = root->right;
            free(root);
            return p;
        }
        else if(root->right == NULL) {
            p = root->left;
            free(root);
            return p;
        }
        /* или есть оба поддерева */
        else {
            p2 = root->right;
            p = root->right;
            while(p->left) p = p->left;
            p->left = root->left;
            free(root);
            return p2;
        }
    }
    if(root->info < key) root->right = dtree(root->right, key);
    else root->left = dtree(root->left, key);
    return root;
}
```

Необходимо также следить за правильным обновлением указателя на корень дерева, описанного вне данной функции, поскольку удаляемая вершина может быть корнем. Лучше всего с этой целью указателю на корень присваивать значение, возвращаемое функцией `dtree()`:

```
■ root = dtree(root, key);
```

Двоичные деревья — исключительно мощное, гибкое и эффективное средство. Поскольку при поиске в сбалансированном дереве выполняется в худшем случае $\log_2 n$ сравнений, оно намного лучше, чем связанный список, в котором возможен лишь последовательный поиск.

Полный
справочник по



Глава 23

Разреженные массивы

Одна из наиболее интересных задач программирования — реализация разреженных массивов. *Разреженный массив*, или *разреженная матрица* (sparse array), — это массив, в котором не все элементы используются, имеются в наличии или нужны в данный момент. Разреженные массивы полезны в тех случаях, когда выполняются два условия: размер массива, который требуется приложению, достаточно большой (возможно, превышает объем доступной памяти), и когда не все элементы массива используются. Таким образом, разреженный массив — это, как правило, большой, но редко заполненный массив. Как будет показано далее, есть несколько способов реализации разреженных массивов. Но перед тем как приступить к их рассмотрению, давайте уделим внимание задачам, которые решаются с помощью разреженных массивов.

Зачем нужны разреженные массивы?

Чтобы понять, зачем нужны разреженные массивы, вспомните следующие два факта:

- При описании обычного массива в языке С вся память, требуемая для размещения массива, выделяется сразу после его создания.
- Большие массивы, особенно многомерные, могут занимать огромные объемы памяти.

Тот факт, что память для массива выделяется при его создании, означает, что размер самого большого массива, который вы сможете описать в своей программе, ограничен (в частности) объемом доступной памяти. Если вам понадобится массив большего размера, чем позволяют возможности компьютера, придется реализовывать его каким-то другим образом. (Например, для работы с полностью заполненными большими массивами обычно применяется та или иная форма виртуальной памяти.) Даже если большой массив разместится в памяти, создание его может существенно уменьшить доступные ресурсы системы, поскольку память, занятая большим массивом, оказывается недоступной для остальной части программы и других процессов, работающих в системе. А это в свою очередь может отрицательно сказаться на общей производительности программы или компьютера в целом. В тех ситуациях, когда будут использоваться не все элементы массива, выделение памяти под весь массив представляется особенно расточительной тратой системных ресурсов.

Чтобы избавиться от проблем, вызванных потребностью в памяти для больших редко заполненных массивов, были придуманы некоторые приемы работы с разреженными массивами. Все они характеризуются одной общей чертой: память для элементов массива выделяется только при необходимости. Поэтому преимущество разреженного массива состоит в том, что для его хранения требуется ровно столько памяти, сколько нужно для хранения только тех элементов, которые действительно используются. При этом остальная память может использоваться для других целей. Кроме того, эти приемы позволяют создавать очень большие массивы, размер которых значительно больше, чем допускаемый системой размер обычных массивов.

Существуют многочисленные примеры приложений, требующих обработки разреженных массивов. Многие из них относятся к работе с матрицами или к научным и инженерным задачам, которые понятны лишь экспертам в соответствующих областях. Однако есть одно очень популярное приложение, в котором обычно применяется разреженный массив — электронная таблица. Несмотря на то, что матрица в средней электронной таблице очень большая, в любой момент времени используется лишь небольшая ее часть. В ячейках электронных таблиц хранятся формулы, значения и строки. При использовании разреженного массива память для каждого элемента выделяется только при необходимости. Поскольку фактически используется лишь небольшая

часть элементов массива, весь он (то есть электронная таблица) кажется очень большим, но занимает минимально необходимую память.

В этой главе будут часто повторяться два термина: *логический массив* и *физический массив*. Логический массив — это массив, который мыслится как существующий в системе. Например, если матрица электронной таблицы имеет размер 1 000×1 000, то логический массив, реализующий матрицу, также будет иметь размер 1 000×1 000 — даже несмотря на то, что физически такой массив не существует в компьютере. Физический массив — это массив, который реально существует в памяти компьютера. Так, если используются только 100 элементов матрицы электронной таблицы, то для хранения физического массива требуется память, необходимая для хранения лишь этих 100 элементов. Методы обработки разреженных массивов, раскрытые в данной главе, обеспечивают связь между логическими и физическими массивами.

Ниже будут рассмотрены четыре различных способа создания разреженного массива: связанный список, двоичное дерево, массив указателей и хеширование. Несмотря на то, что программа работы с электронными таблицами не разрабатывается целиком, все примеры ориентируются на матрицу электронной таблицы, показанную на рис. 23.1. На этом рисунке элемент X расположен в ячейке B2.



Представление разреженного массива в виде связанного списка

При реализации разреженного массива с помощью связанного списка первым делом необходимо создать структуру, содержащую следующие элементы:

- Хранимые в ячейке данные
- Логическая позиция ячейки в массиве
- Ссылки на предыдущий и следующий элементы

	---A---	---B---	---C---
1			
2		X	
3			
4			
5			
6			
7			
.			
.			
.			

Рис. 23.1. Организация простой электронной таблицы

Каждая новая структура помещается в список так, что элементы остаются упорядоченными по индексу в массиве. Доступ к массиву производится путем перехода по ссылкам.

Например, в качестве носителя элемента разреженного массива в электронной таблице можно использовать следующую структуру:

```
struct cell {  
    char cell_name[9]; /* имя ячейки, напр. A1, B34 */
```

```

char formula[128]; /* информация, напр. 10/В2 */
struct cell *next; /* указатель на следующую запись */
struct cell *prior; /* указатель на предыдущую запись */
} ;

```

Поле `cell_name` содержит строку, соответствующую имени ячейки, например, A1, B34 или Z19. Строковое поле `formula` хранит формулу (данные) из соответствующей ячейки таблицы.

Полная программа обработки электронных таблиц была бы слишком большой¹, чтобы использовать ее в качестве примера. Вместо этого в данной главе рассмотрены ключевые функции, обеспечивающие реализацию разреженного массива на основе связанного списка. Следует помнить, что существует очень много способов реализации программы обработки электронных таблиц. Показанные здесь функции и структура данных — лишь примеры приемов работы с разреженными массивами.

Следующие глобальные переменные указывают на начало и конец связанного списка:

```

struct cell *start = NULL; /* первый элемент списка */
struct cell *last = NULL; /* последний элемент списка */

```

В большинстве электронных таблиц при вводе формулы в ячейку создается новый элемент разреженного массива. Если электронная таблица построена на основе связанного списка, этот элемент вставляется в список с помощью функции, аналогичной функции `dls_store()`, приведенной в главе 22. Помните, что список упорядочен по именам ячеек; например, A12 предшествует A13 и т.д.

```

/* Вставка ячеек в упорядоченный список. */
void dls_store(struct cell *i, /* указатель на вставляемую ячейку */
               struct cell **start,
               struct cell **last)
{
    struct cell *old, *p;

    if(!*last) { /* первый элемент в списке */
        i->next = NULL;
        i->prior = NULL;
        *last = i;
        *start = i;
        return;
    }

    p = *start; /* начать с головы списка */

    old = NULL;
    while(p) {
        if(strcmp(p->cell_name, i->cell_name) < 0) {
            old = p;
            p = p->next;
        }
        else {
            if(p->prior) { /* это элемент из середины */
                p->prior->next = i;
                i->next = p;
                i->prior = p->prior;
                p->prior = i;
            }
            return;
        }
    }
}

```

¹ Многие пользователи шутят, что сама Microsoft не знает, сколько же точно занимает ее Excel. Конечно, это только шутка, но лично я иногда думаю, что в ней 100 % правды. — *Прим. ред.*

```

    }
    i->next = p; /* новый первый элемент */
    i->prior = NULL;
    p->prior = i;
    *start = i;
    return;
}
}
old->next = i; /* вставка в конец */
i->next = NULL;
i->prior = old;
*last = i;
return;
}

```

В приведенной выше функции параметр *i* — указатель на новую вставляемую ячейку. Параметры *start* и *last* являются соответственно указателями на указатели на начало и конец списка.

Нижеследующая функция `deletecell()` удаляет из списка ячейку, имя которой передается в качестве параметра.

```

void deletecell(char *cell_name,
                struct cell **start,
                struct cell **last)
{
    struct cell *info;

    info = find(cell_name, *start);
    if(info) {
        if(*start==info) {
            *start = info->next;
            if(*start) (*start)->prior = NULL;
            else *last = NULL;
        }
        else {
            if(info->prior) info->prior->next = info->next;
            if(info != *last)
                info->next->prior = info->prior;
            else
                *last = info->prior;
        }
        free(info); /* освободить системную память */
    }
}

```

Последняя функция, которая понадобится для реализации разреженного массива на основе связанного списка — это функция `find()`, находящая указанную ячейку. Для нахождения ячейки данной функции приходится выполнять линейный поиск, но, как было показано в главе 21, среднее количество сравнений при линейном поиске равно $n/2$, где n — количество элементов в списке. Ниже приведен текст функции `find()`:

```

struct cell *find(char *cell_name, struct cell *start)
{
    struct cell *info;

    info = start;
    while(info) {
        if(!strcmp(cell_name, info->cell_name)) return info;
        info = info->next; /* перейти к следующей ячейке */
    }
}

```

```

    }
    printf("Ячейка не найдена.\n");
    return NULL; /* поиск неудачный */
}

```

Анализ метода представления в виде связанного списка

Принципиальное преимущество метода реализации разреженного массива с помощью связанного списка заключается в том, что он позволяет эффективно использовать память — место выделяется только для тех элементов массива, которые действительно содержат информацию. Кроме того, он прост в реализации. Тем не менее, у этого метода есть один большой недостаток: для доступа к ячейкам в нем применяется линейный поиск. Причем процедура сохранения ячейки также использует линейный поиск, чтобы найти место вставки нового элемента. Эти проблемы можно разрешить, построив разреженный массив на основе двоичного дерева, как показано ниже.



Представление разреженного массива в виде двоичного дерева

По сути, двоичное дерево — это просто видоизмененный двусвязный список. Его основное преимущество заключается в возможности быстрого поиска. Именно благодаря этому удастся очень быстро выполнять вставки и затрачивать совсем немного времени на доступ к элементам. (Ведь двоичные деревья идеально подходят для приложений, в которых требуется структура связанного списка, в которой поиск должен занимать совсем немного времени.)

Чтобы использовать двоичное дерево для реализации электронной таблицы, необходимо изменить структуру `cell` следующим образом:

```

struct cell {
    char cell_name[9]; /* имя ячейки, напр. A1, B34 */
    char formula[128]; /* данные, напр. 10/B2 */
    struct cell *left; /* указатель на левое поддерево */
    struct cell *right; /* указатель на правое поддерево */
} list_entry;

```

Функцию `stree()` из главы 22 можно модифицировать так, чтобы она строила дерево на основании имени ячейки. В следующем коде предполагается, что параметр `n` является указателем на вставляемый элемент дерева.

```

struct cell *stree(
    struct cell *root,
    struct cell *r,
    struct cell *n)
{
    if(!r) { /* первая вершина в поддереве */
        n->left = NULL;
        n->right = NULL;
        if(!root) return n; /* первый вход в дерево */
        if(strcmp(n->cell_name, root->cell_name) < 0)
            root->left = n;
        else
            root->right = n;
        return n;
    }
}

```

```

    if(strcmp(r->cell_name, n->cell_name) <= 0)
        stree(r, r->right, n);
    else
        stree(r, r->left, n);

    return root;
}

```

При вызове функции `stree()` ей необходимо передавать указатели на корень дерева в качестве первых двух параметров и указатель на новую ячейку в качестве третьего. Функция возвращает указатель на корень.

Чтобы удалить ячейку электронной таблицы, можно воспользоваться показанной ниже модифицированной функцией `dtree()`, принимающей в качестве ключа имя ячейки:

```

struct cell *dtree(
    struct cell *root,
    char *key)
{
    struct cell *p, *p2;

    if(!root) return root; /* элемент не найден */

    if(!strcmp(root->cell_name, key)) { /* удаление корня */
        /* это означает пустое дерево */
        if(root->left == root->right) {
            free(root);
            return NULL;
        }
        /* или если одно из поддеревьев пустое */
        else if(root->left == NULL) {
            p = root->right;
            free(root);
            return p;
        }
        else if(root->right == NULL) {
            p = root->left;
            free(root);
            return p;
        }
        /* или если оба поддерева непустые */
        else {
            p2 = root->right;
            p = root->right;
            while(p->left) p = p->left;
            p->left = root->left;
            free(root);
            return p2;
        }
    }
    if(strcmp(root->cell_name, key)<=0)
        root->right = dtree(root->right, key);
    else root->left = dtree(root->left, key);
    return root;
}

```

Наконец, для быстрого поиска ячейки электронной таблицы по ее имени можно воспользоваться модифицированной версией функции `search()`.

```

struct cell *search_tree(
    struct cell *root,

```

```

        char *key)
    {
        if(!root) return root; /* пустое дерево */
        while(strcmp(root->cell_name, key)) {
            if(strcmp(root->cell_name, key) <= 0)
                root = root->right;
            else root = root->left;
            if(root == NULL) break;
        }
        return root;
    }

```

Анализ метода представления в виде двоичного дерева

Применение двоичного дерева значительно уменьшает время вставки и поиска элементов по сравнению со связанным списком. Следует помнить, что последовательный поиск требует в среднем $n/2$ сравнений, где n — количество элементов списка. По сравнению с этим двоичный поиск требует только $\log_2 n$ сравнений (если дерево сбалансировано). Кроме того, двоичные деревья так же экономно расходуют память, как и связанные списки. Тем не менее, в некоторых ситуациях есть лучшие альтернативы, чем двоичные деревья.



Представление разреженного массива в виде массива указателей

Допустим, что электронная таблица имеет размер 26×100 (от A1 до Z100), то есть состоит из 2 600 элементов. Теоретически можно хранить элементы таблицы в следующем массиве структур:

```

struct cell {
    char cell_name[9];
    char formula[128];
} list_entry[2600]; /* 2600 ячеек */

```

Но 2 600, умноженное на 137 байтов (размер этой структуры в байтах), равняется 356 200 байтов памяти. Это слишком большой объем памяти, чтобы тратить его на не полностью используемый массив. Тем не менее, можно создать *массив указателей* (pointer array) на структуры типа cell. Для хранения массива указателей требуется намного меньше памяти, чем для массива структур. При каждом присвоении ячейке логического массива данных под эти данные выделяется память, а соответствующему указателю в массиве указателей присваивается адрес выделенного фрагмента. Такая схема позволяет добиться более высокой производительности, чем при связанном списке или двоичном дереве. Описание массива указателей выглядит следующим образом:

```

struct cell {
    char cell_name[9];
    char formula[128];
} list_entry;

struct cell *sheet[2600]; /* массив из 2600 указателей */

```

Этот меньший по объему занимаемой памяти массив используется для хранения указателей на вводимые в электронную таблицу данные. При вводе очередной записи в соответствующую ячейку массива записывается указатель на введенные данные. На

рис. 23.2 показано, как выглядит в памяти разреженный массив, представленный в виде массива указателей.

Перед использованием массива указателей каждый его элемент необходимо проинициализировать нулем (NULL), что означает отсутствие данной записи. Ниже показана функция инициализации массива:

```
void init_sheet(void)
{
    register int t;

    for(t=0; t < 2600; ++t) sheet[t] = NULL;
}
```

Представление разреженного массива в виде массива указателей

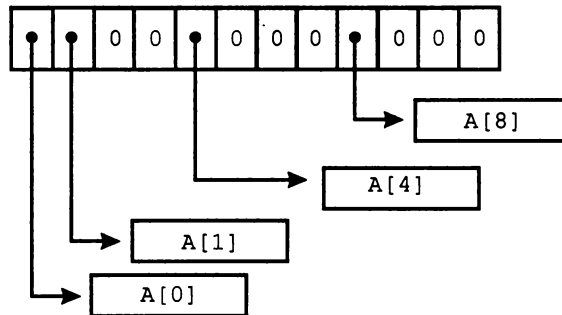


Рис. 23.2. Представление разреженного массива в виде массива указателей

Когда пользователь вводит формулу в ячейку, занимающую определенное положение в электронной таблице (а положение ячейки, как известно, определяется ее именем), вычисляется индекс в массиве указателей `sheet`. Этот индекс получается путем преобразования строкового представления имени ячейки в число, как показано в следующем листинге:

```
void store(struct cell *i)
{
    int loc;
    char *p;

    /* вычисление индекса по заданному имени */
    loc = *(i->cell_name) - 'A'; /* столбец */
    p = &(i->cell_name[1]);
    loc += (atoi(p)-1) * 26; /* количество строк * ширина строки +
                               столбец */

    if(loc >= 2600) {
        printf("Ячейка за пределами массива.\n");
        return;
    }
    sheet[loc] = i; /* поместить указатель в массив */
}
```

При вычислении индекса в функции `store()` предполагается, что все имена ячеек начинаются с прописной буквы, за которой следует целое число, например, B34, C19 и т. д. Поэтому в результате вычислений по формуле, запрограммированной в функции `store()`, имя ячейки A1 соответствует индексу 0, имя B1 соответствует индексу

1, A2 — 26 и т. д. Поскольку имена ячеек уникальны, индексы также уникальны и указатель на каждую запись хранится в соответствующей позиции массива. Если сравнить эту процедуру с версиями, использующими связанный список или двоичное дерево, становится понятно, насколько она проще и короче.

Функция удаления ячейки `deletecell()` также становится очень короткой. При вызове она просто обнуляет указатель на элемент и возвращает системе память.

```
void deletecell(struct cell *i)
{
    int loc;
    char *p;

    /* вычисление индекса по заданному имени ячейки */
    loc = *(i->cell_name) - 'A'; /* столбец */
    p = &(i->cell_name[1]);
    loc += (atoi(p)-1) * 26; /* количество строк * ширина строки +
                               столбец */

    if(loc >= 2600) {
        printf("Ячейка за пределами массива.\n");
        return;
    }
    if(!sheet[loc]) return; /* не освобождать, если указатель нулевой
                              (null) */

    free(sheet[loc]); /* освободить системную память */
    sheet[loc] = NULL;
}
```

Этот код также намного быстрее и проще, чем в версии со связанным списком.

Процесс поиска ячейки по имени прост, поскольку имя ячейки однозначно определяет индекс в массиве указателей. Поэтому функция поиска принимает следующий вид:

```
struct cell *find(char *cell_name)
{
    int loc;
    char *p;

    /* вычисление индекса по заданному имени ячейки */
    loc = *(i->cell_name) - 'A'; /* столбец */
    p = &(i->cell_name[1]);
    loc += (atoi(p)-1) * 26; /* количество строк * ширина строки +
                               столбец */

    if(loc >= 2600 || !sheet[loc]) { /* эта ячейка пустая */
        printf("Ячейка не найдена.\n");
        return NULL; /* поиск неуспешный */
    }
    else return sheet[loc];
}
```

Анализ метода представления разреженного массива в виде массива указателей

Метод реализации разреженного массива на основе массива указателей обеспечивает намного более быстрый доступ к элементам, чем методы на основе связанного списка и двоичного дерева. Если массив не очень большой, выделение памяти для массива указателей лишь незначительно уменьшает объем свободной памяти системы.

Тем не менее, в массиве указателей для каждой ячейки выделяется некоторый объем памяти независимо от того, используется она или нет. В некоторых приложениях это может быть ограничением, хотя в общем случае это не является проблемой.

Хэширование

Хэширование (hashing) — это процесс получения индекса элемента массива непосредственно в результате операций, производимых над ключом, который хранится вместе с элементом или даже совпадает с ним. Генерируемый индекс называется *хэш-адресом* (hash). Традиционно хэширование применяется к дисковым файлам как одно из средств уменьшения времени доступа. Тем не менее, этот общий метод можно применить и с целью доступа к разреженным массивам. В предыдущем примере с массивом указателей использовалась специальная форма хэширования, которая называется *прямая адресация*. В ней каждый ключ соответствует одной и только одной ячейке массива¹. Другими словами, каждый индекс, вычисленный в результате хэширования, уникален. (При представлении разреженного массива в виде массива указателей хэш-функция не должна обязательно реализовывать прямую адресацию — просто это был очевидный подход к реализации электронной таблицы.) В реальной жизни схемы прямого хэширования встречаются редко; обычно требуется более гибкий метод. В данном разделе показано, как можно обобщить метод хэширования, придав ему большую мощь и гибкость.

В примере с электронной таблицей понятно, что даже в самых сложных случаях используются не все ячейки таблицы. Предположим, что почти во всех случаях фактически занятые ячейки составляют не более 10 процентов потенциально доступных мест. Это значит, что если таблица имеет размер 260×100 (2 600 ячеек), в любой момент времени будет использоваться лишь примерно 260 ячеек. Этим подразумевается, что самый большой массив, который понадобится для хранения всех занятых ячеек, будет в обычных условиях состоять только из 260 элементов. Но как ячейки логического массива сопоставить этому меньшему физическому массиву? И что происходит, когда этот массив переполняется? Ниже предлагается одно из возможных решений.

Когда пользователь вводит данные в ячейку электронной таблицы (т.е. заполняет элемент логического массива), позиция ячейки, определяемая по ее имени, используется для получения индекса (хэш-адреса) в меньшем физическом массиве. При выполнении хэширования физический массив называется также *первичным массивом*. Индекс в первичном массиве получается из имени ячейки, которое преобразуется в число, точно так, как и в примере с массивом указателей. Но затем это число делится на 10, в результате чего получается начальная точка входа в первичный массив. (Помните, что в данном случае размер физического массива составляет только 10 % размера логического массива.) Если ячейка физического массива по этому индексу свободна, в нее заносятся логический индекс и данные. Но поскольку 10 логических позиций соответствуют одной физической позиции, могут возникнуть коллизии при вычислении хэш-адресов². Когда это происходит, записи сохраняются в связанном списке, иногда называемом *списком коллизий* (collision list). С каждой ячейкой первичного массива связан отдельный список коллизий. Конечно, до возникновения коллизии эти списки имеют нулевую длину, как показано на рис. 23.3.

Предположим, требуется найти элемент в физическом массиве по его логическому индексу. Сначала необходимо преобразовать логический индекс в соответствующее

¹ Иными словами, хэш-функция является биекцией. — *Прим. ред.*

² Т.е. ситуации, когда разным ключам $k_1 \neq k_2$ соответствует один и тот же хэш-адрес: $h(k_1) = h(k_2)$ (здесь h — хэш-функция). — *Прим. ред.*

значение хэш-адреса и проверить, соответствует ли логический индекс, хранящийся в полученной позиции физического массива, искомому. Если да, информацию можно извлечь. В противном случае необходимо просматривать список коллизий для данной позиции до тех пор, пока не будет найден требуемый логический индекс или пока не будет достигнут конец цепочки.

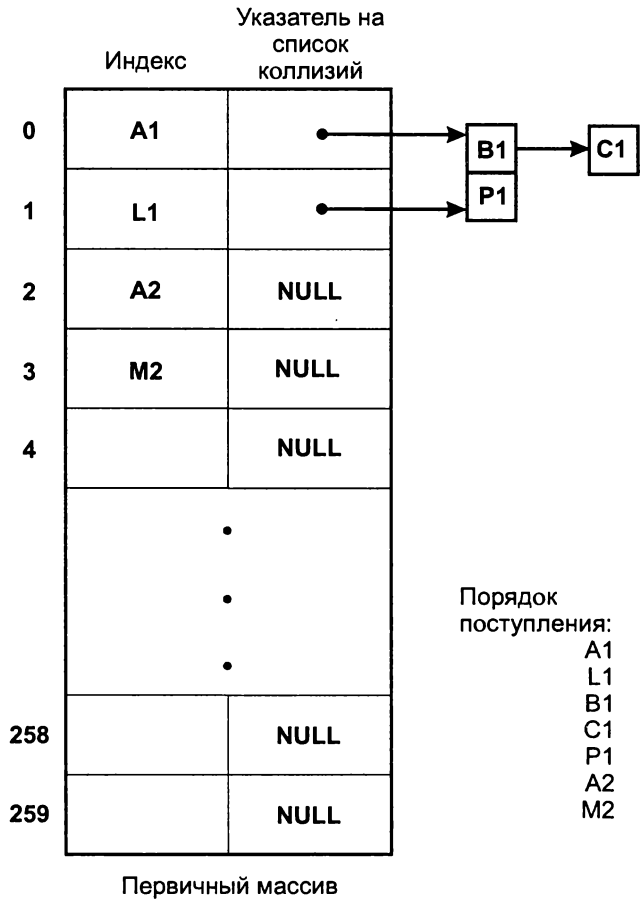


Рис. 23.3. Пример хэширования

В примере хэширования используется массив структур под названием primary:

```
#define MAX 260

struct htype {
    int index; /* логический индекс */
    int val; /* собственно значение элемента данных */
    struct htype *next; /* указатель на следующий элемент с таким же хэш-адресом */
} primary[MAX];
```

Перед использованием этого массива необходимо его инициализировать. Следующая функция присваивает полю index значение -1 (значение, которое по определению нико-

гда не будет сгенерировано в качестве индекса); это значение обозначает пустой элемент. Значение NULL в поле next соответствует пустой цепочке хэширования¹.

```
/* Инициализация хэш-массива. */
void init(void)
{
    register int i;

    for (i=0; i<MAX; i++) {
        primary[i].index = -1;
        primary[i].next = NULL; /* пустая цепочка */
        primary[i].val = 0;
    }
}
```

Процедура store() преобразует имя ячейки в хэш-адрес в первичном массиве primary. Если позиция, на которую указывает значение хэш-адрес, занята, процедура автоматически добавляет запись в список коллизий с помощью модифицированной версии функции slstore() из предыдущей главы. Логический индекс также сохраняется, поскольку он понадобится при извлечении элемента. Данные функции показаны ниже:

```
/* Вычисление хэш-адреса и сохранение значения. */
void store(char *cell_name, int v)
{
    int h, loc;
    struct htype *p;

    /* получение хэш-адреса */
    loc = *cell_name - 'A'; /* столбец */
    loc += (atoi(&cell_name[1])-1) * 26; /* строка * ширина + столбец */
    h = loc/10; /* хэш-адрес */

    /* Сохранить в полученной позиции, если она не занята
    либо если логические индексы совпадают - то есть, при обновлении.
    */
    if(primary[h].index==-1 || primary[h].index==loc) {
        primary[h].index = loc;
        primary[h].val = v;
        return;
    }

    /* в противном случае, создать список коллизий
    либо добавить в него элемент */
    p = (struct htype *) malloc(sizeof(struct htype));
    if(!p) {
        printf("Не хватает памяти\n");
        return;
    }
    p->index = loc;
    p->val = v;
    slstore(p, &primary[h]);
}

/* Добавление элементов в список коллизий. */
void slstore(struct htype *i,
```

¹ Цепочка хэширования (hash chain) — цепочка, соединяющая элементы хэш-таблицы с одним и тем же хэш-кодом. Ранее автор назвал ее *списком коллизий* (collision list). Иногда она называется также *пакетом переполнения*. — Прим. ред.

```

        struct htype *start)
{
    struct htype *old, *p;

    old = start;
    /* найти конец списка */
    while(start) {
        old = start;
        start = start->next;
    }
    /* связать с новой записью */
    old->next = i;
    i->next = NULL;
}

```

Для того чтобы получить значение элемента, программа должна сначала вычислить хэш-адрес и проверить, совпадает ли с искомым логический индекс, хранящийся в полученной позиции физического массива. Если совпадает, возвращается значение ячейки; в противном случае — производится поиск в списке коллизий. Функция `find()`, выполняющая эти задачи, показана ниже:

```

/* Вычисление хэш-адреса и получение значения. */
int find(char *cell_name)
{
    int h, loc;
    struct htype *p;

    /* получение значения хэш-адреса */
    loc = *cell_name - 'A'; /* столбец */
    loc += (atoi(&cell_name[1])-1) * 26; /* строка * ширина + столбец */
    h = loc/10; /* хэш-адрес */

    /* вернуть значение, если ячейка найдена */
    if(primary[h].index == loc) return(primary[h].val);
    else { /* в противном случае просмотреть список коллизий */
        p = primary[h].next;
        while(p) {
            if(p->index == loc) return p->val;
            p = p->next;
        }
        printf("Ячейки нет в массиве\n");
        return -1;
    }
}

```

Создание функции удаления оставлено читателю в качестве упражнения. (Подсказка: Просто обратите процесс вставки.)

Показанный выше алгоритм хэширования очень прост. Как правило, на практике применяются более сложные методы, обеспечивающие более равномерное распределение индексов в первичном массиве, что устраняет длинные цепочки хэширования. Тем не менее, основной принцип остается таким же.

Анализ метода хэширования

В лучшем случае (довольно редком) каждый физический индекс, вычисляемый хэш-функцией, уникален, а время доступа примерно равно времени доступа при прямой адресации. Это значит, что списки коллизий не создаются, а все операции выборки являются по сути операциями прямого доступа. Однако так бывает редко, по-

скольку для этого требуется, чтобы логические индексы равномерно распределялись в пространстве физических индексов. В худшем случае (также редком) схема хэширования вырождается в связанный список. Это происходит, когда значения хэш-адресов всех логических индексов совпадают. В среднем (и наиболее вероятном) случае время доступа при хэшировании равно времени доступа при прямой адресации плюс некоторая константа, пропорциональная средней длине цепочек хэширования. Самый важный фактор при реализации разреженных массивов методом хэширования — выбор такого алгоритма хэширования, при котором равномерно распределяются физические индексы, что позволяет избежать образования длинных списков коллизий.

Выбор метода

При выборе одного из методов представления разреженного массива — с помощью связанного списка, двоичного дерева, массива указателей или хэширования — необходимо руководствоваться требованиями к скорости и эффективному использованию памяти. Кроме того, необходимо учесть, насколько плотно будет заполнен разреженный массив.

Если логический массив заполнен очень редко, самыми эффективными по использованию памяти оказываются связанные списки и двоичные деревья, поскольку в них память выделяется только для тех элементов, которые действительно используются. Для самих связей требуется очень небольшой дополнительный объем памяти, которым обычно можно пренебречь. Представление в виде массива указателей предполагает создание целиком всего массива из указателей, даже если некоторые элементы не используются. При этом в памяти должен не только поместиться весь массив, но еще должна остаться свободная память для работы приложения. В некоторых случаях это может быть серьезной проблемой. Обычно можно заранее оценить примерный объем свободной памяти и решить, достаточен ли он для вашей программы. Метод хэширования занимает промежуточное место между представлениями с помощью массива указателей и связанного списка или двоичного дерева. Несмотря на то, что весь первичный массив, даже если он не используется, должен находиться в памяти, этот массив все равно будет меньше, чем массив указателей.

Если логический массив будет заполнен довольно плотно, ситуация существенно меняется. В этом случае создание массива указателей и хэширование становятся более приемлемыми методами. Более того, время поиска элемента в массиве указателей постоянно и не зависит от степени заполнения логического массива. Хотя время поиска при хэшировании и не постоянно, оно ограничено сверху некоторым небольшим значением. Но в случае связанного списка и двоичного дерева среднее время поиска увеличивается по мере заполнения массива. Это следует помнить, если время доступа является критическим фактором.

Полный
справочник по



Глава 24

**Синтаксический разбор
и вычисление выражений**

Как написать программу, которая будет получать на входе строку, содержащую числовое выражение, например $(10 - 5) * 3$, и выдавать соответствующий результат? Если среди программистов и есть “высшие священники”, то это те, кто знает, как решить подобную задачу. Многие, притом высококвалифицированные в других областях программисты не имеют представления о том, как трансляторы, разработанные для компиляции программ, написанных на языках высокого уровня, преобразовывают алгебраические выражения в команды, выполняемые компьютером. Эта процедура называется *синтаксический разбор выражений* (expression parsing) и является основой всех компиляторов и интерпретаторов языков, электронных таблиц и всех остальных программ, в которых требуется превращать числовые выражения в форму, понятную компьютеру.

Несмотря на свою загадочность, синтаксический разбор выражений является довольно прямолинейным процессом и во многих аспектах проще, чем некоторые другие задачи программирования. Это обусловлено тем, что задача синтаксического разбора четко определена и решается в соответствии со строгими правилами алгебры. В настоящей главе будет разработан *рекурсивный нисходящий синтаксический анализатор*, или *синтаксический анализатор методом рекурсивного спуска* (recursive-descent parser), а также все функции, необходимые для вычисления выражений. Освоив принцип действия этой программы, вы с легкостью сможете доработать и модифицировать ее в соответствии со своими задачами.

На заметку

В интерпретаторе языка C, представленном в части VI данной книги, применяется улучшенный вариант разработанного здесь синтаксического анализатора. Если вы будете изучать интерпретатор C, материал данной главы будет вам особенно полезен.

Выражения

Несмотря на то, что выражения можно составлять из данных любых типов, в настоящей главе рассматриваются только числовые выражения. Для наших целей мы условимся, что числовые выражения будут состоять из следующих элементов:

- Числа
- Операторы (знаки операций) $+$, $-$, $/$, $*$, $^$, $\%$, $=$
- Скобки
- Переменные

Оператор $^$ означает возведение в степень, как в языке BASIC, а символ $=$ обозначает оператор присваивания. Перечисленные элементы можно комбинировать в выражения согласно правилам алгебры, например:

```
10 - 8
(100 - 5) * 14 / 6
a + b - c
10^5
a = 10 - b
```

Пусть операторы имеют следующий приоритет:

высший	унарные $+$ и $-$
	$^$
	$*$ / $\%$
	$+$ -
низший	$=$

Если выражение содержит операторы, имеющие одинаковый приоритет, то вычисления выполняются слева направо¹.

В примерах данной главы все переменные имеют имена из одной буквы (другими словами, допускается 26 переменных, от А до Z). Имена переменных не чувствительны к регистру (заглавных или строчных букв). Например, а и А обозначают одну и ту же переменную. Каждое числовое значение имеет тип double, хотя не составляет труда написать процедуры для обработки значений других типов. Наконец, чтобы логика программ была простой и понятной, будет производиться лишь минимальный контроль за ошибками.

Если вы еще не задумывались о процессе синтаксического разбора выражений, попробуйте вычислить следующее выражение:

$10 - 2 * 3$

Вы знаете, что оно равно 4. Несмотря на то, что можно легко создать программу, которая вычислит данное конкретное выражение, вопрос состоит в том, как написать программу, выдающую правильный результат для произвольного выражения. В начале вам может прийти в голову следующий алгоритм:

```
a = получить первый операнд
while(есть операнды) {
    op = получить оператор
    b = получить второй операнд
    a = a op b
}
```

Эта процедура получает первый операнд, оператор и второй операнд, выполняет над ними первую операцию, а затем читает следующий оператор и операнд (если они есть) и выполняет следующую операцию, обозначенную полученным оператором и т. д.

Однако при данном подходе при вычислении выражения $10 - 2 * 3$ в результате получается 24 (т. е. $8 * 3$) вместо 4, поскольку описанная процедура не учитывает приоритет операторов. Нельзя просто выбирать операнды и операторы слева направо, поскольку правила алгебры гласят, что умножение производится прежде вычитания. Некоторые начинающие программисты думают, что эту проблему легко преодолеть. В очень редких случаях им это удастся. Но проблема усложняется при добавлении скобок, возведении в степень, появлении переменных, вызове функций и т. п.

Несмотря на то, что существует несколько способов написания программы вычисления выражений, описываемый нами способ наиболее прост для кодирования человеком. Он также является самым распространенным. (При некоторых других методах создания синтаксических анализаторов в них применяются сложные таблицы, генерируемые другой компьютерной программой. Такие анализаторы иногда называются *таблично управляемыми* (table-driven).) Описанный здесь метод называется методом *рекурсивного спуска*, и, читая главу, вы, несомненно, догадаетесь, почему он так называется.



Разбиение выражения на лексемы

Для того чтобы вычислять выражения, необходимо уметь разбивать их на отдельные составляющие. Например, выражение $A * B - (W + 10)$ состоит из таких элементов: A, *, B, -, (, W, +, 10 и). Каждый из них представляет единую неделимую часть

¹ Впрочем, здесь есть одно часто встречающееся исключение — оператор возведения в степень. $X^{**}Y^{**}Z$ означает, как правило, не $(X^{**}Y)^{**}Z$, а $X^{**}(Y^{**}Z)$. (Оператор ** — обычное обозначение операции возведения в степень во многих языках программирования, наиболее распространенным из которых является Фортран.) Точно то же происходит и в алгебре: выражение a^{b^c} означает $a^{(b^c)}$, а не $(a^b)^c = a^{bc}$. — Прим. ред.

выражения. В общем случае необходима функция, которая возвращает один за другим все элементы выражения. Эта функция также должна уметь пропускать пробелы и символы табуляции и определять конец выражения.

Каждый элемент выражения называется *лексемой* (token). Поэтому функция, возвращающая очередную лексему, часто называется `get_token()`. В этой функции используется глобальный указатель на строку с разбираемым выражением. В показанной здесь версии функции `get_token()` этот глобальный указатель называется `prog`. Переменная `prog` описана глобально, поскольку она должна сохранять свое значение между вызовами функции `get_token()` и быть доступной другим функциям. Помимо значения возвращаемой лексемы, необходимо знать ее тип. Для анализатора, разрабатываемого в данной главе, понадобятся только три типа: переменная, число и разделитель. Им соответствуют константы `VARIABLE`, `NUMBER` и `DELIMITER`. (`DELIMITER` используется как для операторов, так и для скобок.) Ниже приведен текст функции `get_token()` вместе с необходимыми глобальными описаниями, константами и вспомогательной функцией:

```
#define DELIMITER 1
#define VARIABLE  2
#define NUMBER    3

extern char *prog; /* указатель на анализируемое выражение */
char token[80];
char tok_type;

/* Данная функция возвращает очередную лексему. */
void get_token(void)
{
    register char *temp;

    tok_type = 0;
    temp = token;
    *temp = '\0';

    if(!*prog) return; /* конец выражения */
    while(isspace(*prog)) ++prog; /* пропустить пробелы, символы
                                   табуляции и пустой строки */

    if(strchr("+-*/%^=()", *prog)){
        tok_type = DELIMITER;
        /* продвинуться к следующему символу */
        *temp++ = *prog++;
    }
    else if(isalpha(*prog)) {
        while(!isdelim(*prog)) *temp++ = *prog++;
        tok_type = VARIABLE;
    }
    else if(isdigit(*prog)) {
        while(!isdelim(*prog)) *temp++ = *prog++;
        tok_type = NUMBER;
    }
    *temp = '\0';
}

/* Возвращает значение ИСТИНА, если с является разделителем. */
int isdelim(char c)
{

```

```

    if(strchr(" +-*%^=()", c) || c==9 || c=='\r' || c==0)
        return 1;
    return 0;
}

```

Давайте рассмотрим приведенные выше функции более подробно. После нескольких инициализаций функция `get_token()` проверяет, не достигнут ли символ конца строки ('0'), завершающий выражение. Если в выражении еще есть неразобранная часть, функция `get_token()` сначала пропускает ведущие пробелы, если они имеются. После этого переменная `prog` указывает на число, переменную, оператор или — если выражение завершалось пробелами — на символ конца строки ('0'). Если очередной символ является оператором, он возвращается в виде строки, хранимой в глобальной переменной `token`, а переменной `tok_type`, содержащей тип полученной лексемы, присваивается значение `DELIMITER`. Если же следующий символ является буквой, он считается именем переменной и возвращается в строковой переменной `token`. При этом `tok_type` получает значение `VARIABLE`. В случае, когда очередной символ является цифрой, считывается все число, причем оно помещается в переменную `token`, а его типом будет `NUMBER`. Наконец, если следующий символ не является ни одним из перечисленных выше, считается, что достигнут конец выражения. В этом случае `token` содержит пустую строку, возврат которой означает конец выражения.

Как уже было сказано ранее, чтобы не усложнять код этой функции, были опущены некоторые средства контроля за ошибками и сделаны некоторые допущения. Например, любой нераспознанный символ завершает выражение. Кроме того, в данной версии программы имена переменных могут иметь любую длину, но значащей является только первая буква. В соответствии с требованиями конкретной задачи вы можете усложнить средства контроля за ошибками и добавить другие подробности. Функцию `get_token()` можно доработать или модифицировать, чтобы она выбирала из входного выражения строки символов, числа других типов или лексемы другого типа.

Чтобы лучше понять принцип действия функции `get_token()`, ниже приведены возвращаемые ей лексемы и типы лексем для следующего входного выражения:

$A + 100 - (B * C) / 2$

Лексема	Тип лексемы
A	VARIABLE
+	DELIMITER
100	NUMBER
-	DELIMITER
(DELIMITER
B	VARIABLE
*	DELIMITER
C	VARIABLE
)	DELIMITER
/	DELIMITER
2	NUMBER
нуль (конец строки)	0 (нуль)

Следует помнить, что переменная `token` всегда содержит строку, завершающуюся символом конца строки ('0'), даже если эта строка состоит только из одного символа.

Разбор выражений

Существуют различные способы синтаксического разбора и вычисления выражений. При работе с рекурсивным нисходящим синтаксическим анализатором можно представлять себе выражения в виде *рекурсивных структур данных*, т.е. определение выражения рекурсивно. Иными словами, понятие выражения определяется через понятие выражения. Например, если принять, что в выражениях можно использовать только +, -, *, / и скобки, то все выражения можно определить следующими правилами:

выражение \rightarrow слагаемое [+ слагаемое] [- слагаемое]
слагаемое \rightarrow множитель [* множитель] [/ множитель]
множитель \rightarrow переменная, число или (выражение)

Квадратные скобки означают необязательный элемент, а символ \rightarrow означает “порождает”. Подобные правила обычно называются *порождающими правилами*, или *продукциями*. Поэтому в качестве определения *слагаемого* можно привести следующее: “Слагаемое порождает множитель, умноженный на множитель, или множитель, деленный на множитель”. Обратите внимание на то, что приоритет операций заложен в определении выражения.

Давайте рассмотрим пример. Выражение $10 + 5 * C$ состоит из двух слагаемых: 10 и $5 * C$. Второе слагаемое состоит из двух множителей: 5 и C . Эти множители представляют собой одно число и одну переменную.

С другой стороны, выражение $14 * (7 - C)$ содержит два множителя: 14 и $(7 - C)$. Эти множители представляют собой одно число и одно выражение в скобках. Выражение в скобках состоит из двух слагаемых: числа и переменной.

Описанный процесс анализа выражений составляет основу работы рекурсивного нисходящего синтаксического анализатора, который, по существу, состоит из набора взаимно рекурсивных функций, вызывающих друг друга по цепочке. На каждом этапе своей работы анализатор выполняет указанные операции в алгебраически корректной последовательности. Чтобы увидеть, как это происходит, давайте разберем приведенное ниже выражение в соответствии с вышеуказанными порождающими правилами и выполним арифметические операции:

$$9 / 3 - (100 + 56)$$

Если выражение разобрано корректно, разбор происходил в следующем порядке:

1. Получить первое слагаемое, $9 / 3$.
2. Получить каждый множитель и выполнить деление чисел. В результате получилось число 3.
3. Получить второе слагаемое, $(100 + 56)$. В этот момент начинается рекурсивная обработка данного подвыражения.
4. Получить оба слагаемых и выполнить сложение. В результате получилось число 156.
5. Вернуться из рекурсивного вызова и вычесть 156 из 3. Окончательным ответом является -153.

Если вас это несколько сбilo с толку, не расстраивайтесь. Синтаксический разбор выражений — довольно сложное занятие, к которому нужно привыкнуть. Необходимо помнить о двух основных моментах, когда речь идет о таком рекурсивном представлении выражений. Во-первых, приоритет операций неявно закладывается в порождающие правила. Во-вторых, этот метод разбора и вычисления выражений очень похож на тот способ, которым люди вычисляют математические выражения.



Простая программа синтаксического анализа выражений

В оставшейся части данной главы приведены два синтаксических анализатора. Первый из них разбирает и вычисляет только константные выражения, т.е. выражения, в которых нет переменных. Второй анализатор способен работать с 26 переменными, от A до Z.

Ниже приводится полная версия простого рекурсивного нисходящего синтаксического анализатора, вычисляющего выражения, в которых при вычислении операнды представляются в формате с плавающей запятой.

```
/* Этот модуль содержит простой синтаксический анализатор,
   который не распознает переменные.
*/

#include <stdlib.h>
#include <ctype.h>
#include <stdio.h>
#include <string.h>

#define DELIMITER 1
#define VARIABLE 2
#define NUMBER 3

extern char *prog; /* содержит анализируемое выражение */
char token[80];
char tok_type;

void eval_exp(double *answer), eval_exp2(double *answer);
void eval_exp3(double *answer), eval_exp4(double *answer);
void eval_exp5(double *answer), eval_exp6(double *answer);
void atom(double *answer);
void get_token(void), putback(void);
void serror(int error);
int isdelim(char c);

/* Точка входа анализатора. */
void eval_exp(double *answer)
{
    get_token();
    if(!*token) {
        serror(2);
        return;
    }
    eval_exp2(answer);

    if(*token) serror(0); /* последней лексемой должен быть нуль */
}

/* Сложение или вычитание двух слагаемых. */
void eval_exp2(double *answer)
{
    register char op;
    double temp;
```

```

eval_exp3(answer);
while((op = *token) == '+' || op == '-') {
    get_token();
    eval_exp3(&temp);
    switch(op) {
        case '-':
            *answer = *answer - temp;
            break;
        case '+':
            *answer = *answer + temp;
            break;
    }
}
}

/* Умножение или деление двух множителей. */
void eval_exp3(double *answer)
{
    register char op;
    double temp;

    eval_exp4(answer);
    while((op = *token) == '*' || op == '/' || op == '%') {
        get_token();
        eval_exp4(&temp);
        switch(op) {
            case '*':
                *answer = *answer * temp;
                break;
            case '/':
                if(temp == 0.0) {
                    serror(3); /* деление на 0 */
                    *answer = 0.0;
                } else *answer = *answer / temp;
                break;
            case '%':
                *answer = (int) *answer % (int) temp;
                break;
        }
    }
}

/* Возведение в степень */
void eval_exp4(double *answer)
{
    double temp, ex;
    register int t;

    eval_exp5(answer);

    if(*token == '^') {
        get_token();
        eval_exp4(&temp);
        ex = *answer;
        if(temp == 0.0) {
            *answer = 1.0;
            return;
        }
    }
}

```

```

        for(t=temp-1; t>0; --t) *answer = (*answer) * (double)ex;
    }
}

/* Вычисление унарных операторов + и -. */
void eval_exp5(double *answer)
{
    register char op;

    op = 0;
    if((tok_type == DELIMITER) && *token=='+' || *token == '-') {
        op = *token;
        get_token();
    }
    eval_exp6(answer);
    if(op == '-') *answer = -(*answer);
}

/* Вычисление выражения в скобках. */
void eval_exp6(double *answer)
{
    if((*token == '(')) {
        get_token();
        eval_exp2(answer);
        if(*token != ')')
            serror(1);
        get_token();
    }
    else
        atom(answer);
}

/* Получение значения числа. */
void atom(double *answer)
{
    if(tok_type == NUMBER) {
        *answer = atof(token);
        get_token();
        return;
    }
    serror(0); /* иначе синтаксическая ошибка в выражении */
}

/* Возврат лексемы во входной поток. */
void putback(void)
{
    char *t;

    t = token;
    for(; *t; t++) prog--;
}

/* Отображение сообщения об ошибке. */
void serror(int error)
{
    static char *e[] = {
        "Синтаксическая ошибка",
        "Несбалансированные скобки",
    }
}

```

```

        "Нет выражения",
        "Деление на нуль"
    };
    printf("%s\n", e[error]);
}

/* Возврат очередной лексемы. */
void get_token(void)
{
    register char *temp;

    tok_type = 0;
    temp = token;
    *temp = '\0';

    if(!*prog) return; /* конец выражения */
    while(isspace(*prog)) ++prog; /* пропустить пробелы,
        символы табуляции и пустой строки */

    if(strchr("+-*/%^=()", *prog)){
        tok_type = DELIMITER;
        /* перейти к следующему символу */
        *temp++ = *prog++;
    }
    else if(isalpha(*prog)) {
        while(!isdelim(*prog)) *temp++ = *prog++;
        tok_type = VARIABLE;
    }
    else if(isdigit(*prog)) {
        while(!isdelim(*prog)) *temp++ = *prog++;
        tok_type = NUMBER;
    }

    *temp = '\0';
}

/* Возвращает значение ИСТИНА, если с является разделителем. */
int isdelim(char c)
{
    if(strchr(" +-/*%^=()", c) || c==9 || c=='\r' || c==0)
        return 1;
    return 0;
}

```

В приведенном здесь виде анализатор поддерживает следующие операторы: +, -, *, /, %. Кроме того, он умеет возводить в целочисленную степень (^) и вычислять унарный минус. А еще анализатор умеет корректно распознавать скобки. Обратите внимание, что он состоит из шести уровней, а также функции `atom`, которая возвращает значение числа. Как уже обсуждалось ранее, в глобальной переменной `token` возвращается очередная лексема из строки, содержащей выражение, а в `tok_type` — тип лексемы. Переменная-указатель `prog` указывает на строку, содержащую выражение.

Следующая простая функция `main()` демонстрирует использование этого анализатора:

```

/* Демонстрационная программа для анализатора. */
#include <stdlib.h>
#include <ctype.h>
#include <stdio.h>

```



```

#include <string.h>

char *prog;
void eval_exp(double *answer);

int main(void)
{
    double answer;
    char *p;

    p = (char *) malloc(100);
    if(!p) {
        printf("Ошибка при выделении памяти.\n");
        exit(1);
    }

    /* Обработка выражений до ввода пустой строки. */
    do {
        prog = p;
        printf("Введите выражение: ");
        gets(prog);
        if(!*prog) break;
        eval_exp(&answer);
        printf("Результат: %.2f\n", answer);
    } while(*p);

    return 0;
}

```

Чтобы понять, как же в действительности анализатор вычисляет выражение, давайте проработаем следующий пример. (Допустим, что `prog` указывает на начало выражения.)

10 - 3 * 2

При вызове функции `eval_exp()` — входной точки анализатора — из входной строки выбирается лексема. Если она является пустой строкой, то функция печатает сообщение “Нет выражения” и завершается. Однако в данном случае лексемой является число 10. Поскольку это не пустая строка, вызывается функция `eval_exp2()`. В результате, функция `eval_exp2()` вызывает `eval_exp3()`, а `eval_exp3()` вызывает `eval_exp4()`, а та в свою очередь вызывает `eval_exp5()`. Затем функция `eval_exp5()` проверяет, не является ли лексема унарным плюсом или минусом. В данном случае это не так, поэтому вызывается функция `eval_exp6()`. В этот момент `eval_exp6()` может рекурсивно вызвать либо `eval_exp2()` (в случае выражения, заключенного в скобки), либо `atom()`, чтобы определить значение числа. Поскольку лексема не является открывающей скобкой, выполняется функция `atom()` и переменной `*answer` присваивается значение 10. Затем происходит выборка следующей лексемы и возврат из цепочки вызовов функций. Лексемой становится оператор `-`, а управление возвращается функции `eval_exp2()`.

То, что происходит дальше, очень важно. Поскольку текущей лексемой является символ `-`, он сохраняется в переменной `op`. Затем анализатор выбирает следующую лексему и спуск по цепочке начинается снова. Как и раньше, вызывается функция `atom()`. Полученное значение 3 возвращается в переменной `*answer`, и считывается лексема `*`. Это вызывает возврат по цепочке до `eval_exp3()`, где считывается последняя лексема 2. В этот момент происходит первая арифметическая операция — умножение 3 на 2. Полученный результат возвращается функции `eval_exp2()`, где происходит вычитание. В результате вычитания в ответе получается 4. Несмотря на

то, что этот процесс может сначала показаться сложным, самостоятельная проработка других примеров поможет вам разобраться в работе анализатора.

Данный анализатор подошел бы для настольного калькулятора, что было продемонстрировано предыдущей программой, или для небольшой базы данных. Однако перед тем как использовать его для разбора языка программирования или в сложном калькуляторе, в него необходимо добавить средства работы с переменными. Это является предметом следующего раздела.

Работа с переменными в анализаторе

Во всех языках программирования, многих калькуляторах и электронных таблицах предусмотрены переменные, позволяющие сохранять значения для дальнейшего использования. Для того чтобы синтаксический анализатор из предыдущего примера обладал такой возможностью, в него необходимо внести некоторые дополнения. Во-первых, это, конечно, сами переменные. Как уже было сказано выше, анализатор будет распознавать только переменные с именами от A до Z. (Впрочем, при желании вы можете избавиться от этого ограничения.) Каждая переменная хранится в одной ячейке массива из 26 элементов типа `double`. Поэтому в исходный текст анализатора необходимо добавить следующий фрагмент:

```
double vars[26] = { /* 26 пользовательских переменных,  A-Z */
    0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
    0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
    0.0, 0.0, 0.0, 0.0, 0.0, 0.0 };
```

Как вы заметили, для удобства пользователя все переменные инициализируются нулями.

Кроме этого, понадобится процедура для получения значения заданной переменной. Поскольку имена переменных являются буквами от A до Z, их можно использовать для индексации массива `vars`, вычитая код ASCII буквы A из имени переменной. Ниже показана функция `find_var()`, возвращающая значение переменной:

```
/* Получение значения переменной. */
double find_var(char *s)
{
    if(!isalpha(*s)){
        error(1);
        return 0;
    }
    return vars[toupper(*token) - 'A'];
}
```

Данная функция написана так, что она принимает имена любой длины, но только первый символ является значимым. Данное ограничение можно изменить в соответствии с вашими потребностями.

Также необходимо модифицировать функцию `atom()`, чтобы она обрабатывала как числа, так и переменные. Ниже показана ее новая версия:

```
/* Получение значения числа или переменной. */
void atom(double *answer)
{
    switch(tok_type) {
        case VARIABLE:
            *answer = find_var(token);
            get_token();
```

```

        return;
    case NUMBER:
        *answer = atof(token);
        get_token();
        return;
    default:
        serror(0);
}
}

```

С технической точки зрения, это все, что требуется анализатору для корректной обработки переменных. Однако пока нет способа присвоить этим переменным значения. Часто это делается за пределами анализатора, но в анализаторе можно рассматривать знак равенства как знак операции присваивания и сделать обработку этого знака частью анализатора. Этого можно достичь несколькими способами. Один из них — добавить в анализатор функцию `eval_exp1()`, показанную ниже:

```

/* Обработка присваивания. */
void eval_exp1(double *result)
{
    int slot, tok_type;
    char temp_token[80];

    if(tok_type == VARIABLE) {
        /* сохранить старую лексему */
        strcpy(temp_token, token);
        tok_type = tok_type;

        /* вычислить индекс переменной */
        slot = toupper(*token) - 'A';

        get_token();
        if(*token != '=') {
            putback(); /* вернуть текущую лексему */
            /* восстановить старую лексему - это не присваивание */
            strcpy(token, temp_token);
            tok_type = tok_type;
        }
        else {
            get_token(); /* получить следующую часть выражения */
            eval_exp2(result);
            vars[slot] = *result;
            return;
        }
    }

    eval_exp2(result);
}

```

Как вы видите, этой функции приходится заглядывать вперед, чтобы определить, выполняется ли на самом деле присваивание. Это связано с тем, что имя переменной всегда находится перед оператором присваивания, но само по себе наличие имени переменной не гарантирует, что за ней последует присваивание. Другими словами, анализатор воспримет выражение `A = 100` как присваивание, причем он может определить, что `A / 10` им не является. Для этого функция `eval_exp1()` считывает из входного потока следующую лексему. Если эта лексема не является знаком равенства, она с помощью функции `putback()` возвращается во входной поток для последующего использования:

```

/* Возврат лексемы во входной поток. */
void putback(void)
{
    char *t;

    t = token;
    for(; *t; t++) prog--;
}

```

Ниже приведен полный текст улучшенного анализатора:

```

/* Данный модуль содержит рекурсивный нисходящий
   синтаксический анализатор, распознающий переменные.
*/

#include <stdlib.h>
#include <ctype.h>
#include <stdio.h>
#include <string.h>

#define DELIMITER 1
#define VARIABLE 2
#define NUMBER 3

extern char *prog; /* указатель на анализируемое выражение */
char token[80];
char tok_type;

double vars[26] = { /* 26 пользовательских переменных, A-Z */
    0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
    0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
    0.0, 0.0, 0.0, 0.0, 0.0, 0.0
};

void eval_exp(double *answer), eval_exp2(double *answer);
void eval_exp1(double *result);
void eval_exp3(double *answer), eval_exp4(double *answer);
void eval_exp5(double *answer), eval_exp6(double *answer);
void atom(double *answer);
void get_token(void), putback(void);
void serror(int error);
double find_var(char *s);
int isdelim(char c);

/* Точка входа анализатора. */
void eval_exp(double *answer)
{
    get_token();
    if(!*token) {
        serror(2);
        return;
    }
    eval_exp1(answer);
    if(*token) serror(0); /* последняя лексема должна быть нулем */
}

/* Обработка присваивания. */
void eval_exp1(double *answer)
{

```

```

int slot;
char tok_type;
char temp_token[80];

if(tok_type == VARIABLE) {
    /* сохранить старую лексему */
    strcpy(temp_token, token);
    tok_type = tok_type;
    /* вычислить индекс переменной */
    slot = toupper(*token) - 'A';

    get_token();
    if(*token != '=') {
        putback(); /* вернуть текущую лексему */
        /* восстановить старую лексему - это не присваивание */
        strcpy(token, temp_token);
        tok_type = tok_type;
    }
    else {
        get_token(); /* получить следующую часть выражения */
        eval_exp2(answer);
        vars[slot] = *answer;
        return;
    }
}
eval_exp2(answer);
}

/* Сложение или вычитание двух слагаемых. */
void eval_exp2(double *answer)
{
    register char op;
    double temp;

    eval_exp3(answer);
    while((op = *token) == '+' || op == '-') {
        get_token();
        eval_exp3(&temp);
        switch(op) {
            case '-':
                *answer = *answer - temp;
                break;
            case '+':
                *answer = *answer + temp;
                break;
        }
    }
}

/* Умножение или деление двух множителей. */
void eval_exp3(double *answer)
{
    register char op;
    double temp;

    eval_exp4(answer);
    while((op = *token) == '*' || op == '/' || op == '%') {
        get_token();

```

```

eval_exp4(&temp);
switch(op) {
    case '*':
        *answer = *answer * temp;
        break;
    case '/':
        if(temp == 0.0) {
            serror(3); /* деление на нуль */
            *answer = 0.0;
        } else *answer = *answer / temp;
        break;
    case '%':
        *answer = (int) *answer % (int) temp;
        break;
}
}
}

/* Возведение в степень */
void eval_exp4(double *answer)
{
    double temp, ex;
    register int t;

    eval_exp5(answer);
    if(*token == '^') {
        get_token();
        eval_exp4(&temp);
        ex = *answer;
        if(temp==0.0) {
            *answer = 1.0;
            return;
        }
        for(t=temp-1; t>0; --t) *answer = (*answer) * (double)ex;
    }
}

/* Вычисление унарного + и -. */
void eval_exp5(double *answer)
{
    register char op;

    op = 0;
    if((tok_type == DELIMITER) && *token=='+' || *token == '-') {
        op = *token;
        get_token();
    }
    eval_exp6(answer);
    if(op == '-') *answer = -(*answer);
}

/* Обработка выражения в скобках. */
void eval_exp6(double *answer)
{
    if((*token == '(')) {
        get_token();
        eval_exp2(answer);
        if(*token != ')')

```

```

        serror(1);
        get_token();
    }
    else atom(answer);
}

/* Получение значения числа или переменной. */
void atom(double *answer)
{
    switch(tok_type) {
        case VARIABLE:
            *answer = find_var(token);
            get_token();
            return;
        case NUMBER:
            *answer = atof(token);
            get_token();
            return;
        default:
            serror(0);
    }
}

/* Возврат лексемы во входной поток. */
void putback(void)
{
    char *t;

    t = token;
    for(; *t; t++) prog--;
}

/* Отображение сообщения о синтаксической ошибке. */
void serror(int error)
{
    static char *e[] = {
        "Синтаксическая ошибка",
        "Несбалансированные скобки",
        "Нет выражения",
        "Деление на нуль"
    };
    printf("%s\n", e[error]);
}

/* Получение очередной лексемы. */
void get_token(void)
{
    register char *temp;

    tok_type = 0;
    temp = token;
    *temp = '\0';

    if(!*prog) return; /* конец выражения */

    while(isspace(*prog)) ++prog; /* пропустить пробелы,
        символы табуляции и пустой строки */

```

```

if(strchr("+-*/%^=()", *prog)){
    tok_type = DELIMITER;
    /* перейти к следующему символу */
    *temp++ = *prog++;
}
else if(isalpha(*prog)) {
    while(!isdelim(*prog)) *temp++ = *prog++;
    tok_type = VARIABLE;
}
else if(isdigit(*prog)) {
    while(!isdelim(*prog)) *temp++ = *prog++;
    tok_type = NUMBER;
}

*temp = '\0';
}

/* Возвращает значение ИСТИНА, если c является разделителем. */
int isdelim(char c)
{
    if(strchr(" +-*/%^=()", c) || c==9 || c=='\r' || c==0)
        return 1;
    return 0;
}

/* Получение значения переменной. */
double find_var(char *s)
{
    if(!isalpha(*s)){
        error(1);
        return 0.0;
    }
    return vars[toupper(*token) - 'A'];
}

```

Для демонстрации работы данного анализатора можно использовать ту функцию `main()`, которая использовалась для демонстрации работы простого анализатора. Усовершенствованный анализатор позволяет вводить выражения, подобные следующим:

```

A = 10 / 4
A - B
C = A * (F - 21)

```



Проверка синтаксиса в рекурсивном нисходящем анализаторе

При разборе выражений синтаксическая ошибка — это просто ситуация, в которой входное выражение не соответствует строгим правилам анализатора. В большинстве случаев это происходит из-за ошибки человека — обычно из-за опечаток. Например, следующие выражения не являются правильными с точки зрения анализаторов, рассмотренных в данной главе:

```

10 ** 8
(10 - 5) * 9
/ 8

```


В первом из них встречаются два оператора подряд, во втором не сбалансированы скобки, а последнее начинается со знака деления. Ни одна из таких последовательностей не допускается рассмотренными анализаторами. Поскольку при наличии синтаксических ошибок анализатор может выдать неправильный результат, необходимо следить, чтобы подобных ошибок не было.

При изучении кода анализаторов вы, вероятно, заметили функцию `error()`, вызываемую в определенных ситуациях. Эта функция сообщает об ошибках. В отличие от многих других типов анализаторов, рекурсивный спуск облегчает проверку синтаксиса, поскольку в большинстве случаев она происходит в функциях `atom()`, `find_var()` и `eval_exp6()`, где выполняется проверка правильной расстановки скобок. Единственная проблема, связанная с выявлением синтаксических ошибок, заключается в том, что при обнаружении ошибки разбор выражения не прекращается. Это может привести к выводу нескольких сообщений об ошибках.

Лучший способ реализации функции `error()` — заставить ее выполнять нечто вроде восстановления правильного состояния анализатора. Например, все современные компиляторы поставляются вместе с парой вспомогательных функций `setjmp()` и `longjmp()`. Эти функции позволяют осуществить в программе передачу управления из одной функции в *другую*. Например, функция `error()` могла бы выполнять длинный переход с помощью `longjmp()` в безопасную точку программы за пределами анализатора.

Если вы оставите код анализатора без изменений, могут выводиться сразу несколько сообщений о синтаксических ошибках. Конечно, в одних ситуациях это может мешать, но в других может быть очень полезным, поскольку появляется возможность выявить сразу несколько ошибок. Тем не менее, в общем случае перед тем как использовать анализатор в коммерческих программах, необходимо доработать его блок синтаксического контроля.

Полный
справочник по



Глава 25

**Решение задач с помощью
искусственного интеллекта**

Хотя искусственный интеллект (ИИ) как область знаний в последнее время является предметом увлекательного изучения в самых разных аспектах, но в основе большинства его приложений все же лежит решение задач. В сущности, задачи эти делятся на два вида. Задачи первого вида можно решить с помощью какой-либо детерминированной процедуры, которая гарантирует достижение успеха — другими словами, путем простого *вычисления*. Методы решения таких задач часто легко переводятся в алгоритмы, выполняемые на компьютере. Однако в повседневной жизни мало найдется таких задач, решение которых сводится к простому вычислению. В действительности большинство задач относится, наоборот, ко второму виду, т.е. к нечисловым задачам, для решения которых как раз и применяется *поиск решения* — метод, разработанный с помощью искусственного интеллекта.

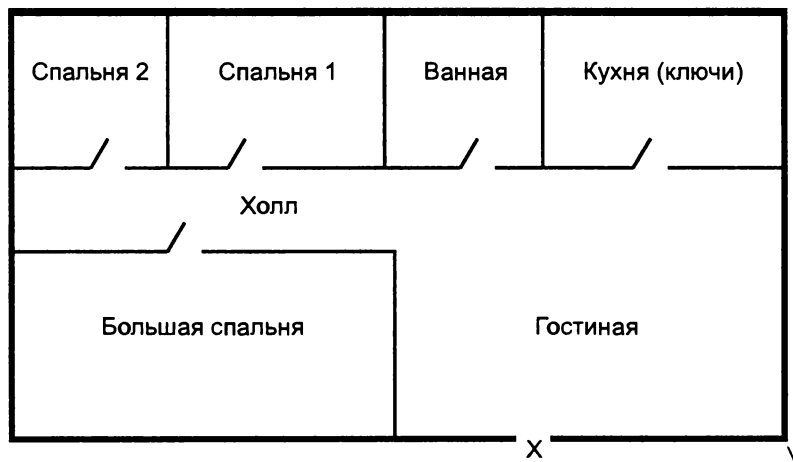
Одной из целей искусственного интеллекта является создание *универсального решателя задач* (*general problem solver, GPS*). Универсальным решателем задач называется программа, которая может находить решения любых задач, не располагая специфическими знаниями в областях, к которым относятся решаемые задачи. В этой главе рассказывается о преимуществах и недостатках программ, близких к универсальным решателям задач.

Первоначально исследования по искусственному интеллекту в основном были нацелены на разработку хороших методов поиска. Тому имелось две причины: необходимость и желание. Чаще всего препятствием для использования приемов искусственного интеллекта в решении повседневных задач является огромный объем данных, а также сложность задач. Для решения этих задач требуются хорошие методы поиска. Кроме того, на начальном этапе исследователи верили, да и сейчас верят, что при решении задач главным является поиск решения и что именно он является решающим компонентом интеллекта.



Представление и терминология

Представьте, что вы потеряли ключи от своей машины. Известно, что они находятся где-то в вашем доме, план которого выглядит примерно так:



Вы стоите там, где находится входная дверь (указанная буквой X). Начиная поиск, вы проверяете гостиную. Потом проходите через холл в первую спальню, затем, опять пересекая холл, — во вторую спальню, возвращаетесь в холл и проходите в большую спальню. Не найдя ключи, вы возвращаетесь, снова проходя через гостиную, и находите свои ключи на кухне. Такую ситуацию легко представить в виде графа (рис. 25.1).

Тот факт, что задачи поиска можно представить в виде графа, является достаточно важным, потому что граф наглядно показывает, как работают разные приемы поиска¹. (Кроме того, возможность представлять задачи в виде графов позволяет исследователям искусственного интеллекта применять разные теоремы теории графов. Однако эти теоремы в книге не рассматриваются.) Помня о такой возможности, познакомьтесь с такими определениями из теории графов:

Вершина (графа)

Лист

Область поиска

Цель

Эвристика

Цепочка, ведущая к решению (путь к решению)

Дискретная точка

Вершина дерева, не имеющая дочерних вершин

Множество всех вершин

Разыскиваемая вершина

Процедура предпочтения при выборе вершины

Оrientированный граф с вершинами, через которые пришлось пройти при поиске решения

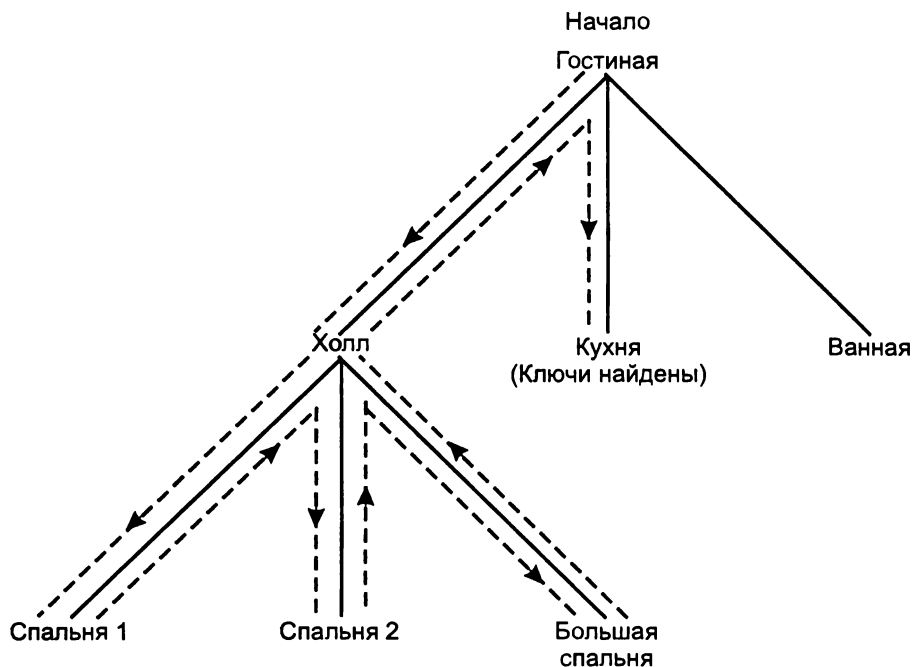


Рис. 25.1. Цепочка, ведущая к решению при поиске потерянных ключей

В примере с потерянными ключами каждая комната в доме — это вершина, весь дом — это область поиска; целью, при достижении которой поиск завершается, является кухня, а цепочка, ведущая к решению, показана на рис. 25.1. Спальни, кухня и ванная являются листьями, потому что никуда дальше не ведут. Хотя в приведенном примере эвристика не используется, но в данной главе мы рассмотрим ее чуть позже.

¹ Поиск связан с методами исследования древовидных структур, с помощью которых представляется предметная область. Поэтому особое значение в методах искусственного интеллекта уделяется поиску в специальных графах — деревьях. По этой же причине и терминология в этой области испытывает сильное влияние теории деревьев. — Прим. ред.

Комбинаторные взрывы

Сейчас вы, возможно, думаете, что поиск решения в данном случае не представляет особого труда — надо лишь методично искать с самого начала и до конца. В этом крайне простом случае с потерянными ключами это не такой уж и плохой метод. Но в процессе поиска решения большинства задач складывается совсем другая ситуация. Обычно компьютер используется для решения задач, в которых количество вершин в области поиска очень большое, а по мере роста области поиска растет и число возможных путей к цели. Проблема состоит в том, что при добавлении новой вершины в области поиска появляется больше чем один новый путь. Значит, количество потенциальных цепочек, ведущих к решению (т.е. путей к цели), растет быстрее, чем количество вершин.

Например, проанализируем количество способов расположения трех объектов: А, В и В. Вот шесть возможных перестановок:

А	В	В
А	В	В
В	В	А
В	А	В
В	В	А
В	А	В

Вы можете легко убедиться, что это все перестановки множества из трех объектов А, В и В. Однако чтобы подсчитать число перестановок, не обязательно выписывать их, достаточно вспомнить математику, точнее одну из первых теорем *комбинаторики*; в комбинаторике, как вы помните, изучаются конечные множества. В самом начале комбинаторики доказывается, что число перестановок множества из N элементов равняется $N!$ (N факториал). Факториал числа — это произведение всех натуральных чисел, расположенных между самим этим числом и 1. Например, $3!$ равен $3 \times 2 \times 1$, то есть 6. Число перестановок четырехэлементного множества равно $4!$, т.е. 24. Для множества из пяти элементов это число равняется 120, а для множества из шести элементов — уже 720. Количество перестановок 1000 элементов равно

402	387	260	077	093	773	543	702	433	923	003	985	719	374	864	210
714	632	543	799	910	429	938	512	398	629	020	592	044	208	486	969
610	197	196	058	631	666	872	994	808	558	901	323	829	669	944	590
073	759	918	823	627	727	188	732	519	779	505	950	995	276	120	874
601	418	278	094	646	496	291	056	393	887	437	886	487	337	119	181
849	977	012	476	632	889	835	955	735	432	513	185	323	958	463	075
417	474	349	347	553	428	646	576	611	667	797	396	668	820	291	207
588	249	808	126	867	838	374	559	731	746	136	085	379	534	524	221
090	878	297	308	431	392	844	403	281	231	558	611	036	976	801	357
609	675	871	348	312	025	478	589	320	767	169	132	448	426	236	131
000	261	683	151	027	341	827	977	704	784	635	868	170	164	365	024
264	810	213	092	761	244	896	359	928	705	114	964	975	419	909	342
080	821	333	186	116	811	553	615	836	546	984	046	708	975	602	900
847	728	421	889	679	646	244	945	160	765	353	408	198	901	385	442
319	101	723	355	556	602	139	450	399	736	280	750	137	837	615	307
034	352	625	200	015	888	535	147	331	611	702	103	968	175	921	510
178	114	194	545	257	223	865	541	461	062	892	187	960	223	838	971
862	967	146	674	697	562	911	234	082	439	208	160	153	780	889	893
671	616	762	179	168	909	779	911	903	754	031	274	622	289	988	005
012	187	361	745	992	642	956	581	746	628	302	955	570	299	024	324
465	832	036	786	906	117	260	158	783	520	751	516	284	225	540	265
143	974	286	933	061	690	897	968	482	590	125	458	327	168	226	458
652	682	272	807	075	781	391	858	178	889	652	208	164	348	344	825
660	176	999	612	831	860	788	386	150	279	465	955	131	156	552	036
138	558	600	301	435	694	527	224	206	344	631	797	460	594	682	573

432	438	465	657	245	014	402	821	885	252	470	935	190	620	929	023	136	493	273	497
565	513	958	720	559	654	228	749	774	011	413	346	962	715	422	845	862	377	387	538
230	483	865	688	976	461	927	383	814	900	140	767	310	446	640	259	899	490	222	221
765	904	339	901	886	018	566	526	485	061	799	702	356	193	897	017	860	040	811	889
729	918	311	021	171	229	845	901	641	921	068	884	387	121	855	646	124	960	798	722
908	519	296	819	372	388	642	614	839	657	382	291	123	125	024	186	649	353	143	970
137	428	531	926	649	875	337	218	940	694	281	434	118	520	158	014	123	344	828	015
051	399	694	290	153	483	077	644	569	099	073	152	433	278	288	269	864	602	789	864
321	139	083	506	217	095	002	597	389	863	554	277	196	742	822	248	757	586	765	752
344	220	207	573	630	569	498	825	087	968	928	162	753	848	863	396	909	959	826	280
956	121	450	994	871	701	244	516	461	260	379	029	309	120	889	086	942	028	510	640
182	154	399	457	156	805	941	872	748	998	094	254	742	173	582	401	063	677	404	595
741	785	160	829	230	135	358	081	840	096	996	372	524	230	560	855	903	700	624	271
243	416	909	004	153	690	105	933	983	835	777	939	410	970	027	753	472	000	000	000
000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000
000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000
000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000
000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000

Поистине колоссально!

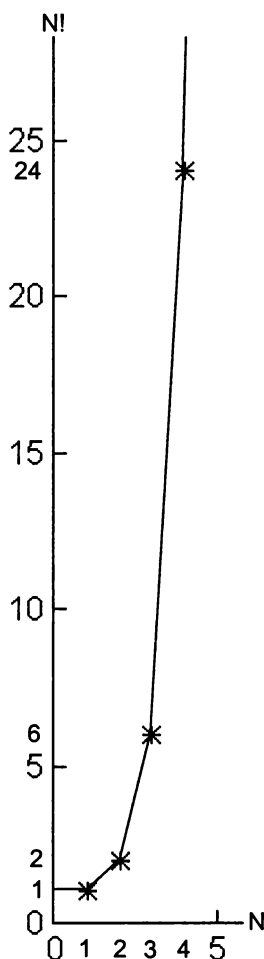


Рис. 25.2. Комбинаторный взрыв, происходящий с факториалами

График на рис. 25.2 дает возможность получить наглядное представление о том, что исследователи искусственного интеллекта называют *комбинаторным взрывом*. И как только количество объектов превысит какое-то сравнительно небольшое число, рост количества комбинаторных объектов (например, путей в графе), перебираемых в процессе решения, становится поистине неудержимым; трудности могут возникнуть даже не при проверке такого огромного количества объектов, а гораздо раньше — при пересчете. Ведь каждая дополнительная вершина в области поиска увеличивает число возможных решений не на 1, а на число, значительно большее 1. Поэтому после достижения некоторого критического количества объектов, добавление еще одного объекта к исходным данным, например, новой вершины, приводит к тому, что возможных “кандидатов в решение” становится так много, что проверить их за обозримое время практически невозможно. Именно из-за того, что количество возможностей растет столь быстро, лишь в самых простых задачах можно применять такую “роскошь”, как *исчерпывающий поиск*. Исчерпывающим называется поиск, при котором проверяются все вершины; этот вид поиска можно считать приемом “грубой силы”. Хотя прием “грубой силы” теоретически применим всегда, на практике он часто требует слишком много времени или слишком много компьютерных ресурсов, или того и другого вместе. Поэтому исследователи разработали другие методы поиска.



Методы поиска

Для поиска решения применяется несколько методов¹. Вот четыре самых главных:

- в глубину;
- в ширину²;
- восходящий³;
- с использованием частичного пути минимальной стоимости.

О каждом из них рассказывается в этой главе.

¹ Поскольку поиск решения часто выполняется путем *перебора*, то термины *поиск* и *перебор* в искусственном интеллекте часто рассматриваются как взаимозаменяемые. А поскольку поиск связан с методами исследования графов, с помощью которых представляется предметная область, то *поиск* (или *перебор*) выполняется с помощью *обхода* графов. Так как в качестве графов чаще всего выступают древовидные структуры, то в процессе перебора выполняется *обход дерева*. Поэтому едва ли стоит удивляться, что вводимые далее термины практически заимствованы из теории обхода деревьев. — *Прим. ред.*

² Называемый также *полным перебором* или *полным поиском*. — *Прим. ред.*

³ Называемый также методом *наискорейшего подъема*, *наискорейшего спуска* и *нисходящим*. Довольно странная терминология, если в ней отождествляются противоположные понятия (*наискорейший подъем* = *наискорейший спуск*, *восходящий* = *нисходящий*), не правда ли? Все дело в том, как растут деревья. Если мне будет позволено так выразиться, я скажу, что согласно учебникам ботаники деревья, конечно же, в основном растут *снизу вверх*, т.е. *корень* находится *внизу*, а *листья* — *вверху*. В компьютерах (и учебниках по информатике) то ли гравитация не является решающим фактором при определении места расположения корней и листьев, то ли программисты (и авторы учебников по информатике) умеют поворачивать вектор гравитации на 180°. Как бы то ни было, но в большинстве учебников по информатике корни деревьев располагаются *вверху* страницы, а сами деревья растут *вниз* (а куда же им в таком случае расти?). Поэтому при обходе деревьев от корня к листьям приходится двигаться *вниз*. Если же корень расположить *внизу*, то дерево будет расти *вверх* (а куда же ему в таком случае расти?). Но в этом случае при обходе деревьев от корня к листьям приходится двигаться *вверх*. Сам алгоритм обхода дерева, естественно, не зависит от направления, в котором растет дерево. Вот и получается, что один и тот же алгоритм называется (разными авторами) по-разному, т.е. получается, что его название зависит от направления, в котором растет дерево. Если хотите, добавьте сюда немного сомнительной философии и получите равенство *вверх=вниз*. — *Прим. ред.*

Оценка поиска

Иногда бывает очень сложно оценить, насколько эффективен метод поиска. Фактически оценка методов поиска и составляет значительную часть искусственного интеллекта. Впрочем, для нас наиболее важными являются два критерия: 1) насколько быстро при поиске находится решение; 2) насколько хорошим является найденное решение.

Имеется несколько видов задач, в процессе решения которых особенно важным является первый из двух критериев, т.е. главным аспектом здесь является то, чтобы решение, возможно, даже любое из решений, было найдено с минимальными усилиями. Однако в других ситуациях решение обязательно должно быть хорошим, возможно, даже оптимальным.

Быстрота поиска определяется как длиной пути решения, так и количеством вершин, через которые фактически приходится пройти в процессе поиска решения. Следует помнить, что при возврате из тупика усилия на самом деле оказываются потраченными впустую. Поэтому необходимо выработать такую стратегию поиска, благодаря которой к минимуму сводится возможность попадания в тупик.

Необходимо понимать разницу между нахождением хорошего и оптимального решения. Чтобы найти оптимальное решение, может потребоваться исчерпывающий поиск, так как иногда именно он является единственным способом проверки того, что было найдено наилучшее решение. А найти хорошее решение — это означает найти такое решение, которое удовлетворяет набору ограничений; при этом не важно, имеется ли решение, которое еще лучше.

Как будет видно из дальнейшего изложения, методы поиска, описанные в этой главе, не во всех ситуациях работают одинаково хорошо. Поэтому трудно сказать, *всегда* ли какой-либо метод лучше другого. Впрочем, “в среднем” некоторые методы, будут более приемлемыми, чем остальные. Кроме того, иногда сам способ постановки задачи подсказывает подходящий метод поиска.

Теперь давайте проанализируем задачу, для решения которой воспользуемся разными методами поиска. Представьте, что вы транспортный агент, а какой-то довольно придирчивый клиент хочет заказать у вас билет от Нью-Йорка до Лос-Анджелеса, причем на самолет именно компании XYZ Airlines. Вы пытаетесь объяснить клиенту, что у этой компании беспересадочных авиарейсов из Нью-Йорка в Лос-Анджелес нет, но он хочет лететь только на самолетах компании XYZ Airlines. Самолеты компании XYZ Airlines по расписанию совершают следующие авиарейсы:

Авиарейс	Расстояние, в милях
Нью-Йорк — Чикаго	1000
Чикаго — Денвер	1000
Нью-Йорк — Торонто	800
Нью-Йорк — Денвер	1900
Торонто — Калгари	1500
Торонто — Лос-Анджелес	1800
Торонто — Чикаго	500
Денвер — Урбана	1000
Денвер — Хьюстон	1500
Хьюстон — Лос-Анджелес	1500
Денвер — Лос-Анджелес	1000

Вы быстро понимаете, что из Нью-Йорка в Лос-Анджелес добраться самолетами компании XYZ Airlines можно только в том случае, если заказать билеты на несколько промежуточных авиарейсов, что вы и делаете.

Ваша задача состоит в том, чтобы написать несколько С-программ, которые будут выбирать маршрут лучше, чем это получается у вас.

■ Представление в виде графа

Информацию об авиарейсах компании XYZ Airlines можно представить в виде ориентированного графа (рис. 25.3). *Ориентированным графом* (или *орграфом*) называется граф, в котором каждое ребро (линия, соединяющая вершины графа) рассматривается как направленное. На рисунке это направление движения по ребру, изображаемому линией, показывает стрелка. В этом графе по ребру нельзя двигаться в направлении, противоположном тому, которое показано его стрелкой.

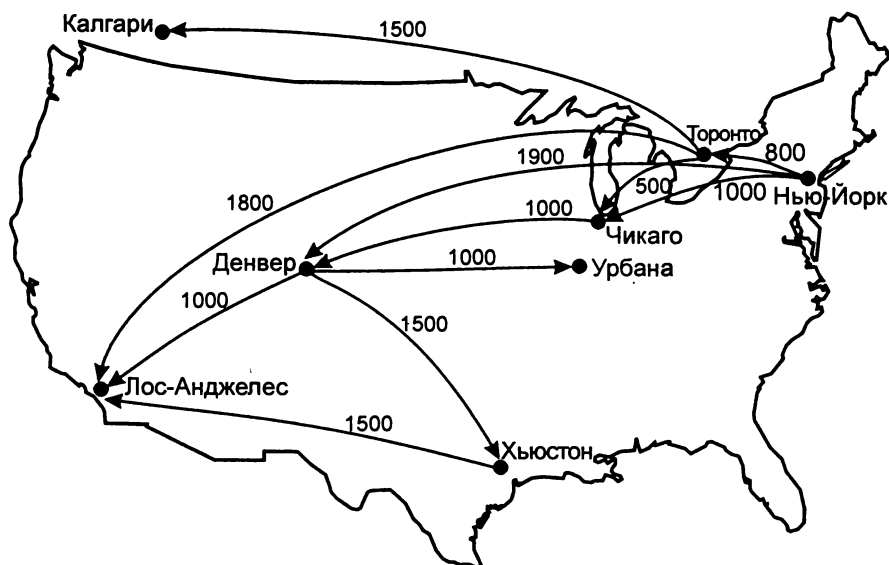


Рис. 25.3. Ориентированный (и даже нагруженный — с весами на ребрах) граф авиарейсов компании XYZ Airlines

Чтобы дать иное, более понятное представление графа авиарейсов компании XYZ Airlines, его изобразили в виде дерева (рис. 25.4). Теперь этот вариант будет использоваться нами вплоть до конца главы. Цель, т.е. конечный пункт путешествия — Лос-Анджелес, — обведена кружком. Кроме того, обратите внимание, что для того чтобы граф было проще представить в виде дерева, некоторые города в нем показаны несколько раз.

Теперь можно приступить к разработке разных программ поиска, с помощью которых можно будет выбирать маршрут от Нью-Йорка до Лос-Анджелеса.

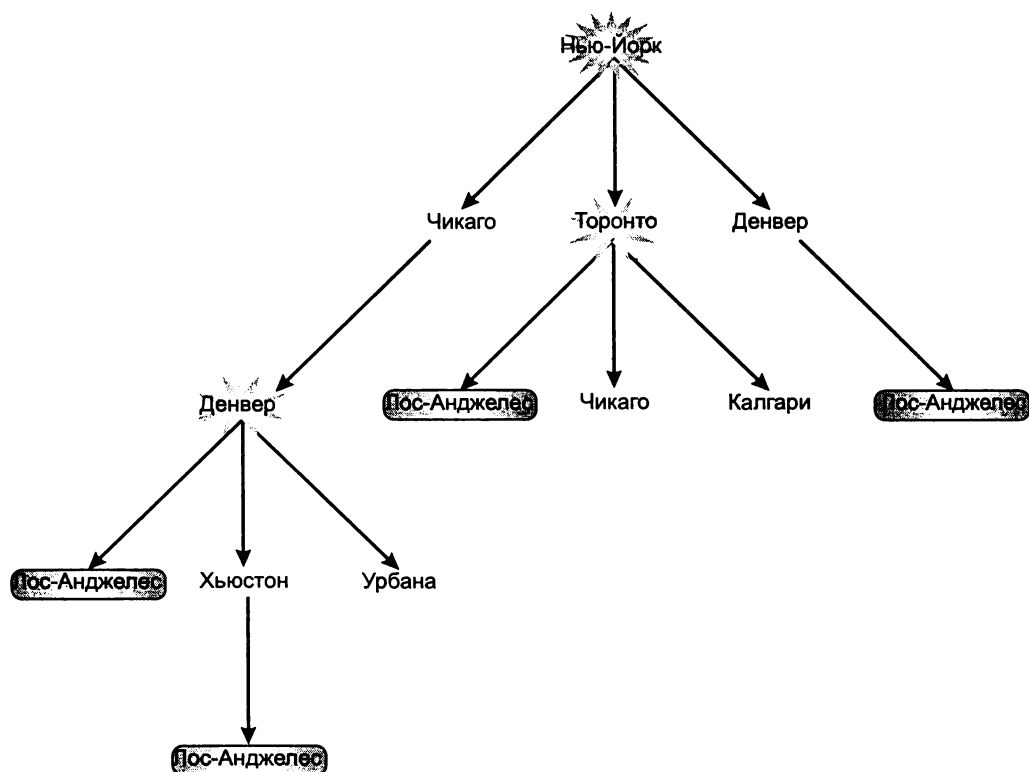
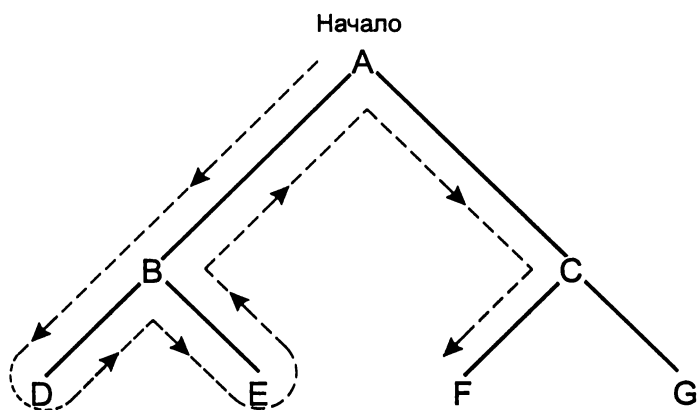


Рис. 25.4. Авиарейсы компании XYZ Airlines , показанные в виде дерева

Поиск в глубину

При *поиске в глубину* каждая из цепочек, которые могут привести к решению, проверяется до своего листа, а лишь затем начинается проверка следующей цепочки (т.е. пути). Чтобы более точно представить себе, как работает такой поиск, проанализируйте следующее дерево. Целью является вершина F.



При поиске в глубину вершины графа обходятся в порядке ABDVEBACF. Если вы знакомы с деревьями, то увидите, что в этом виде поиска используется разновидность обхода неориентированного дерева. То есть путь продолжается налево до тех пор, пока не будет достигнут лист или не будет найдена цель. Если лист достигнут, то путь поворачивает назад, поднимается на один уровень вверх, затем продолжается направо, потом снова влево, пока не встретится цель или очередной лист. А вся эта процедура продолжается до тех пор, пока не будет найдена цель или не проверена последняя вершина области поиска.

Как видите, поиск в глубину, — это такой метод поиска цели, который в наихудшем случае вырождается в исчерпывающий поиск. В нашем примере это случится тогда, когда целью является вершина G.

В С-программе, предназначенной для выбора маршрута из Нью-Йорка в Лос-Анджелес, используется база данных с информацией об авиарейсах компании XYZ Airlines. Каждая запись этой базы содержит сведения о городе-пункте вылета и городе-пункте прибытия, о расстоянии между ними, а также флаг, который помогает при поиске с возвратом (вы скоро увидите, каким именно образом). Вся эта информация хранится в следующей структуре:

```
#define MAX 100

/* структура базы данных авиарейсов */
struct FL {
    char from[20];
    char to[20];
    int distance;
    char skip; /* используется при поиске с возвратом */
};

struct FL flight[MAX]; /* массив структур БД */

int f_pos = 0; /* количество записей в БД авиарейсов */
int find_pos = 0; /* индекс для поиска в БД авиарейсов */
```

Отдельные записи вводятся в базу данных с помощью функции `assert_flight()`, а всю информацию об авиарейсах инициализирует функция `setup()`. Индекс последней записи базы данных сохраняется в глобальной переменной `f_pos`. Вот код нужных нам функций:

```
void setup(void)
{
    assert_flight("Нью-Йорк", "Чикаго", 1000);
    assert_flight("Чикаго", "Денвер", 1000);
    assert_flight("Нью-Йорк", "Торонто", 800);
    assert_flight("Нью-Йорк", "Денвер", 1900);
    assert_flight("Торонто", "Калгари", 1500);
    assert_flight("Торонто", "Лос-Анджелес", 1800);
    assert_flight("Торонто", "Чикаго", 500);
    assert_flight("Денвер", "Урбана", 1000);
    assert_flight("Денвер", "Хьюстон", 1500);
    assert_flight("Хьюстон", "Лос-Анджелес", 1500);
    assert_flight("Денвер", "Лос-Анджелес", 1000);
}

/* Записать данные в базу. */
void assert_flight(char *from, char *to, int dist)
{
    if(f_pos < MAX) {
        strcpy(flight[f_pos].from, from);
```

```

        strcpy(flight[f_pos].to, to);
        flight[f_pos].distance = dist;
        flight[f_pos].skip = 0;
        f_pos++;
    }
    else printf("База данных авиарейсов переполнена.\n");
}

```

Чтобы следовать духу науки об искусственном интеллекте (ИИ), считайте, что в базе данных содержатся “факты”. Создаваемая программа будет с помощью этих “фактов” приближаться к решению. По этой причине многие исследователи искусственного интеллекта называют базы данных “базами знаний”. В данной главе эти понятия используются как взаимозаменяемые.

Для написания кода, выполняющего поиск маршрута из Нью-Йорка в Лос-Анджелес, потребуется несколько вспомогательных функций. Во-первых, нужна подпрограмма для определения того, имеется ли авиарейс между двумя городами или нет. Эта функция называется `match()`; она возвращает расстояние между двумя городами, если такой авиарейс есть и ноль, если такого рейса нет. Вот эта функция:

```

/* Если между двумя городами имеется авиарейс, то возвращается
   расстояние между ними, а в противном случае возвращается 0. */
int match(char *from, char *to)
{
    register int t;

    for(t=f_pos-1; t > -1; t--)
        if(!strcmp(flight[t].from, from) &&
            !strcmp(flight[t].to, to)) return flight[t].distance;

    return 0; /* не найден */
}

```

Другой необходимой подпрограммой является `find()`. Эта функция ищет в базе данных какой-либо авиарейс из указанного города. Если авиарейс с каким-либо другим городом найден, то возвращаются название этого города и расстояние до него от первого города, в противном же случае возвращается ноль. Вот текст подпрограммы `find()`:

```

/* Зная пункт отправления (параметр from), найти пункт прибытия
   (параметр anywhere). */
int find(char *from, char *anywhere)
{
    find_pos = 0;
    while(find_pos < f_pos) {
        if(!strcmp(flight[find_pos].from, from) &&
            !flight[find_pos].skip) {
            strcpy(anywhere, flight[find_pos].to);
            flight[find_pos].skip = 1; /* активизировать */
            return flight[find_pos].distance;
        }
        find_pos++;
    }
    return 0;
}

```

Как видите, если поле `skip` (пропустить) имеет значение 1, то авиарейс между городами не выбирается. Кроме того, когда авиарейс найден, то его поле `skip` отмечается как активное — таким образом реализуется поиск с возвратом из тупиков.

Поиск с возвратом — это очень важная часть многих методов, используемых в системах искусственного интеллекта. Он выполняется с помощью рекурсивных подпрограмм и

с помощью специального стека для поиска с возвратом. Почти во всех ситуациях, связанных с возвратом к предыдущему состоянию, используется нечто, похожее на стек, — то есть нечто, работающее по принципу “последним пришел — первым вышел”. При прохождении по пути все встретившиеся на нем вершины помещаются в стек. При достижении листа, наоборот, происходит как бы остановка без возможности повторного пуска (в глубину) и из стека извлекается последняя помещенная туда вершина и исследуется новый путь, который начинается с этой вершины. Этот процесс продолжается до тех пор, пока не будет найдена цель или не будут исследованы все пути. Далее представлены функции `push()` и `pop()` (означают соответственно “поместить в стек, положить в стек, сохранить в стеке” и “вытаскивать из стека, снимать со стека, вынимать из стека, считывать из стека”), которые управляют стеком, используемым для поиска с возвратом. Массив, в котором хранятся значения, затапливаемые в стек, представлен глобальной переменной `bt_stack`, а указатель на вершину стека — глобальной переменной `tos`. Именно эти переменные используют данные функции при обращении к стеку.

```
/* Подпрограммы обращения к стеку */
void push(char *from, char *to, int dist)
{
    if(tos < MAX) {
        strcpy(bt_stack[tos].from, from);
        strcpy(bt_stack[tos].to, to);
        bt_stack[tos].dist = dist;
        tos++;
    }
    else printf("Стек заполнен.\n");
}

void pop(char *from, char *to, int *dist)
{
    if(tos > 0) {
        tos--;
        strcpy(from, bt_stack[tos].from);
        strcpy(to, bt_stack[tos].to);
        *dist = bt_stack[tos].dist;
    }
    else printf("Стек пуст.\n");
}
```

Теперь, когда написаны необходимые вспомогательные подпрограммы, проанализируйте следующий код. Он определяет функцию `isflight()` — главную подпрограмму для поиска маршрута из Нью-Йорка в Лос-Анджелес.

```
/* Определить, имеется ли маршрут между из города, на который указы-
   вает параметр from (из) в город, на который указывает параметр
   to (в). */
void isflight(char *from, char *to)
{
    int d, dist;
    char anywhere[20];

    /* посмотреть, является ли городом-пунктом прибытия */
    if(d=match(from, to)) {
        push(from, to, d);
        return;
    }

    /* проверить другой авиарейс */
    if(dist=find(from, anywhere)) {
```

```

    push(from, to, dist);
    isflight(anywhere, to);
}
else if(tos > 0) {
    /* поиск с возвратом */
    pop(from, to, &dist);
    isflight(from, to);
}
}

```

Эта подпрограмма работает следующим образом. Во-первых, функция `match()` проверяет базу данных — нет ли в ней авиарейса между городами, на которые указывают параметры `from` и `to`. Если такой авиарейс имеется, то цель поиска достигнута — данные по авиарейсу помещаются в стек и функция возвращает управление. В противном случае функция `find()` проверяет, нет ли авиарейса между городом, на который указывает `from`, и каким-нибудь другим городом. Если есть, то соответствующие данные помещаются в стек и рекурсивно вызывается `isflight()`. В противном случае выполняется поиск с возвратом. Предыдущая вершина удаляется из стека, и рекурсивно вызывается `isflight()`. Этот процесс повторяется до тех пор, пока не будет найдена цель. Поле `skip` используется при поиске с возвратом, чтобы не проверялись повторно одни и те же авиарейсы.

Поэтому если подпрограмму вызвать с такими параметрами, как “Денвер” и “Хьюстон”, то первое условие оператора `if` будет истинным и `isflight()` завершит свою работу. Предположим теперь, что `isflight()` вызывается с параметрами “Чикаго” и “Хьюстон”. В таком случае первое условие оператора `if` не будет истинным, так как авиарейса между этими городами нет. Поэтому второе условие оператора `if` проверяет наличие авиарейса из Чикаго в какой-либо другой город. В нашем примере из Чикаго имеется авиарейс в Денвер, поэтому `isflight()` рекурсивно вызывается с параметрами “Денвер” и “Хьюстон”. И опять проверяется первое условие. Оно выполнено: на этот раз авиарейс найден! Наконец, все рекурсивные вызовы можно завершить, и `isflight()` заканчивает свою работу. Проанализируйте — используемая здесь функция `isflight()` ведет в базе знаний поиск в глубину.

Важно понять, что `isflight()` на самом деле *не возвращает* решение — она его *генерирует*. При выходе из функции `isflight()` в стеке, используемом для поиска с возвратом, находится готовый маршрут между Чикаго и Хьюстоном, то есть собственное решение. В действительности успешное или неудачное выполнение `isflight()` определяется состоянием стека. Пустой стек указывает на неудачу; в противном же случае в стеке будет находиться решение. Поэтому для завершения всей программы нужна еще одна функция. Эта функция называется `route()`, именно она выводит как путь, так и общее расстояние. Рассмотрим эту функцию:

```

/* Вывести маршрут и общее расстояние. */
void route(char *to)
{
    int dist, t;

    dist = 0;
    t = 0;
    while(t < tos) {
        printf("%s - ", bt_stack[t].from);
        dist += bt_stack[t].dist;
        t++;
    }
    printf("%s\n", to);
    printf("Расстояние в милях равно %d.\n", dist);
}

```

Далее следует вся программа, использующая поиск в глубину:

```
/* Поиск в глубину. */
#include <stdio.h>
#include <string.h>

#define MAX 100

/* структура базы данных авиарейсов */
struct FL {
    char from[20];
    char to[20];
    int distance;
    char skip; /* используется при поиске с возвратом */
};

struct FL flight[MAX]; /* массив структур БД */

int f_pos = 0; /* количество записей в БД авиарейсов */
int find_pos = 0; /* индекс для поиска в БД авиарейсов */
int tos = 0; /* вершина стека */
struct stack {
    char from[20];
    char to[20];
    int dist;
} ;
struct stack bt_stack[MAX]; /* стек, используемый для поиска
                             с возвратом */

void setup(void), route(char *to);
void assert_flight(char *from, char *to, int dist);
void push(char *from, char *to, int dist);
void pop(char *from, char *to, int *dist);
void isflight(char *from, char *to);
int find(char *from, char *anywhere);
int match(char *from, char *to);

int main(void)
{
    char from[20], to[20];

    setup();

    printf("Пункт вылета: ");
    gets(from);
    printf("Пункт прибытия: ");
    gets(to);

    isflight(from, to);
    route(to);

    return 0;
}

/* Инициализация базы данных авиарейсов. */
void setup(void)
{
    assert_flight("Нью-Йорк", "Чикаго", 1000);
    assert_flight("Чикаго", "Денвер", 1000);
    assert_flight("Нью-Йорк", "Торонто", 800);
}
```

```

assert_flight("Нью-Йорк", "Денвер", 1900);
assert_flight("Торонто", "Калгари", 1500);
assert_flight("Торонто", "Лос-Анджелес", 1800);
assert_flight("Торонто", "Чикаго", 500);
assert_flight("Денвер", "Урбана", 1000);
assert_flight("Денвер", "Хьюстон", 1500);
assert_flight("Хьюстон", "Лос-Анджелес", 1500);
assert_flight("Денвер", "Лос-Анджелес", 1000);
}

/* Запомнить данные в базе. */
void assert_flight(char *from, char *to, int dist)
{
    if(f_pos < MAX) {
        strcpy(flight[f_pos].from, from);
        strcpy(flight[f_pos].to, to);
        flight[f_pos].distance = dist;
        flight[f_pos].skip = 0;
        f_pos++;
    }
    else printf("База данных авиарейсов заполнена.\n");
}

/* Показать маршрут и общее расстояние. */
void route(char *to)
{
    int dist, t;

    dist = 0;
    t = 0;
    while(t < tos) {
        printf("%s - ", bt_stack[t].from);
        dist += bt_stack[t].dist;
        t++;
    }
    printf("%s\n", to);
    printf("Расстояние в милях равно %d.\n", dist);
}

/* Если между двумя городами имеется авиарейс, то возвращается
   расстояние между ними, в противном случае возвращается 0. */
int match(char *from, char *to)
{
    register int t;

    for(t=f_pos-1; t > -1; t--)
        if(!strcmp(flight[t].from, from) &&
            !strcmp(flight[t].to, to)) return flight[t].distance;

    return 0; /* не найден */
}

/* Зная пункт отправления (параметр from), найти пункт прибытия
   (параметр anywhere). */
int find(char *from, char *anywhere)
{
    find_pos = 0;

```



```

while(find_pos < f_pos) {
    if(!strcmp(flight[find_pos].from,from) &&
        !flight[find_pos].skip) {
        strcpy(anywhere,flight[find_pos].to);
        flight[find_pos].skip = 1; /* активизировать */
        return flight[find_pos].distance;
    }
    find_pos++;
}
return 0;
}

/* Определить, имеется ли маршрут между из города, на который указывает
   параметр from (из) в город, на который указывает параметр to (в). */
void isflight(char *from, char *to)
{
    int d, dist;
    char anywhere[20];

    /* является ли пунктом прибытия */
    if(d=match(from, to)) {
        push(from, to, d);
        return;
    }
    /* проверить другой авиарейс */
    if(dist=find(from, anywhere)) {
        push(from, to, dist);
        isflight(anywhere, to);
    }
    else if(tos > 0) {
        /* поиск с возвратом */
        pop(from, to, &dist);
        isflight(from, to);
    }
}

/* Подпрограммы обращения к стеку */
void push(char *from, char *to, int dist)
{
    if(tos < MAX) {
        strcpy(bt_stack[tos].from,from);
        strcpy(bt_stack[tos].to,to);
        bt_stack[tos].dist = dist;
        tos++;
    }
    else printf("Стек заполнен.\n");
}

void pop(char *from, char *to, int *dist)
{
    if(tos > 0) {
        tos--;
        strcpy(from,bt_stack[tos].from);
        strcpy(to,bt_stack[tos].to);
        *dist = bt_stack[tos].dist;
    }
    else printf("Стек пуст.\n");
}

```

Обратите внимание, что `main()` подсказывает ввести пункт вылета и пункт прибытия. Это означает, что программу можно использовать для определения маршрутов между любыми двумя городами. Однако в оставшейся части главы при анализе программы подразумевается, что вылет осуществляется из Нью-Йорка, а в качестве пункта назначения выбран Лос-Анджелес.

Если выбраны Нью-Йорк и Лос-Анджелес, то получится такое решение:

Нью-Йорк - Чикаго - Денвер - Лос-Анджелес
Расстояние в милях равно 3000.

Цепочка, ведущая к решению и являющаяся собственно маршрутом, показана на рис. 25.5.

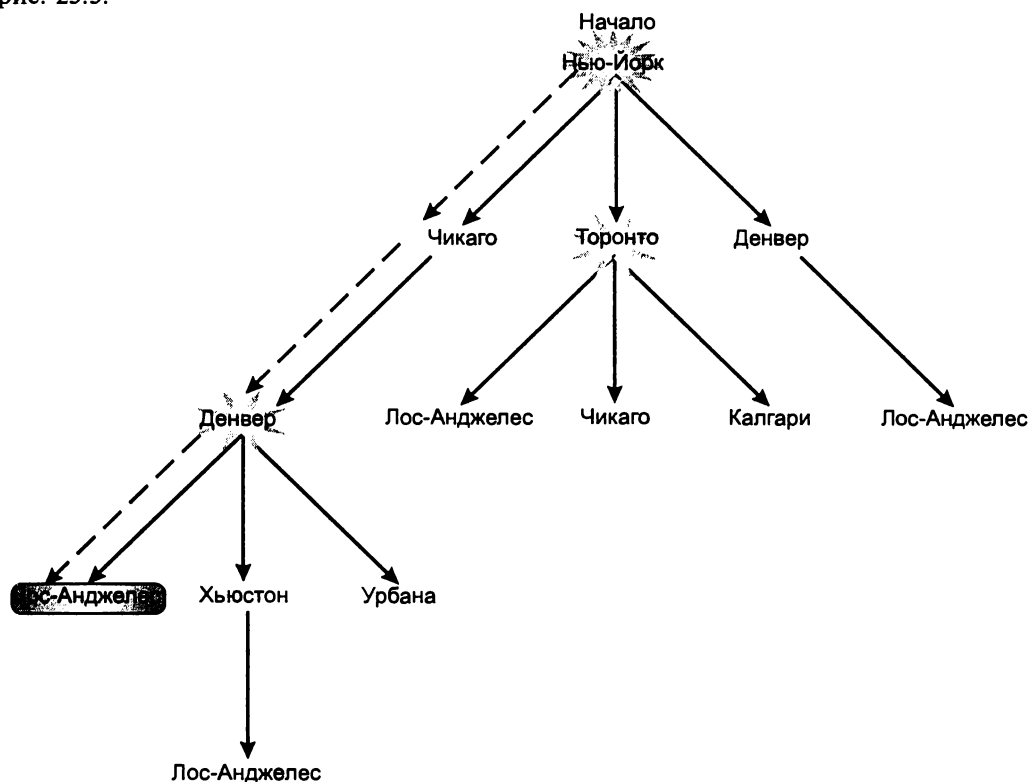


Рис. 25.5. Это решение (цепочка, ведущая к решению, т.е. собственно путь) было найдено с помощью поиска в глубину

На рис. 25.5 видно, что это действительно первое решение, которое можно найти с помощью поиска в глубину. Найденное нами решение не оптимально (оптимальное решение в данном случае — это маршрут Нью-Йорк — Торонто — Лос-Анджелес с расстоянием в 2600 миль), но плохим его тоже назвать нельзя.

Анализ поиска в глубину

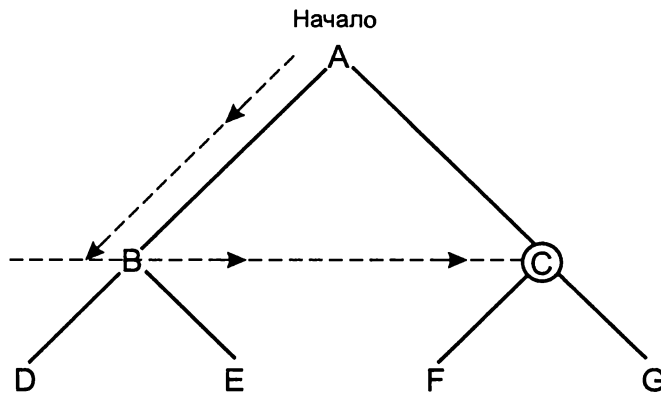
Итак, с помощью поиска в глубину нам удалось найти достаточно хорошее решение. Кроме того, благодаря такому поиску именно в этой задаче было с первой попытки найдено решение без возврата, что уже считается очень хорошим показателем. Но то, что для достижения оптимального решения при поиске в глубину приходится пройти почти все вершины, — вот это уже не совсем хороший признак.

Обратите внимание, что если изучается особенно длинная ветвь, на конце которой нет решения, то эффективность поиска в глубину может оказаться весьма низкой. Тогда при рассматриваемом способе поиска теряется очень много времени, причем не только на изучение этой цепи, но и на возвращение по ней, чтобы можно было двигаться дальше к цели.



Полный перебор, или поиск в ширину

Противоположностью поиска в глубину, является *поиск в ширину*, или *полный перебор*. В соответствии с этим методом вначале обходятся все вершины, находящиеся на одном и том же уровне, а лишь затем выполняется переход на следующий, более низкий уровень. Вот как используется этот метод при поиске цели C:



Чтобы программа поиска маршрута выполняла поиск в ширину, необходимо изменить лишь подпрограмму `isflight()`:

```
void isflight(char *from, char *to)
{
    int d, dist;
    char anywhere[20];

    while(dist=find(from, anywhere)) {
        /* модификация: поиск в ширину */
        if(d=match(anywhere, to)) {
            push(from, to, dist);
            push(anywhere, to, d);
            return;
        }
    }
    /* проверить любой авиарейс */
    if(dist=find(from, anywhere)) {
        push(from, to, dist);
        isflight(anywhere, to);
    }
    else if(tos>0) {
        pop(from, to, &dist);
        isflight(from, to);
    }
}
```

Как можно видеть, изменено только первое условие. Теперь проверяются все города, в которые можно попасть авиарейсом из пункта вылета, но из которых нет авиарейса в пункт прибытия.

Если этой версией `isflight()` заменить в программе предыдущую реализацию данной функции, то получится следующее решение:

■ Нью-Йорк – Торонто – Лос-Анджелес
Расстояние в милях равно 2600.

Это решение является оптимальным. Путь к решению, найденный с помощью поиска в ширину, показан на рис. 25.6.

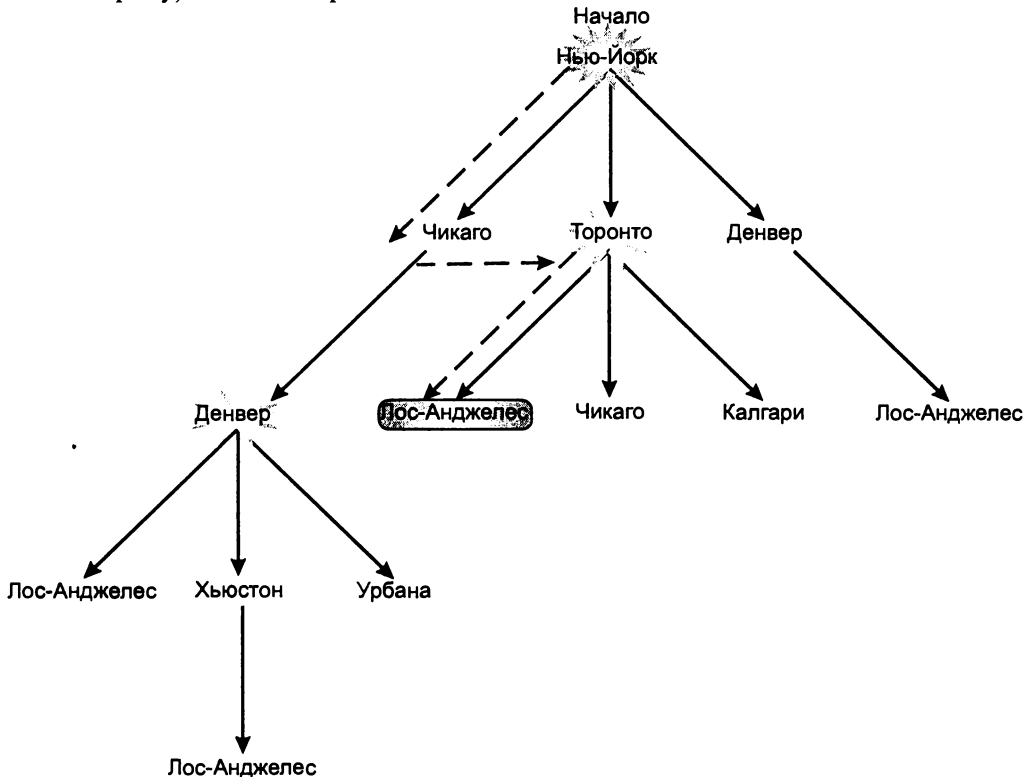


Рис. 25.6. Путь к решению, найденный с помощью поиска в ширину

Анализ поиска в ширину

В этом примере поиск в ширину находит первое решение без возврата. Более того, оказывается, что это решение еще и оптимальное. В действительности первые три решения, которые могли бы быть найдены, как раз и являлись бы самыми лучшими маршрутами. Однако этот результат нельзя обобщить на другие случаи, потому что сгенерированный путь зависит от физической организации хранения информации в компьютере. Зато этот пример хорошо показывает радикальное отличие двух методов поиска: в глубину и в ширину.

Недостатки поиска в ширину становятся очевидными, когда цель находится на глубине нескольких слоев. В таком случае для поиска цели приходится затрачивать значительные усилия. В общем, выбирая один из двух методов поиска, в глубину или в ширину, приходится делать обоснованные догадки о том, где вероятнее всего находится цель.

Добавление эвристики

К этому времени вы, возможно, догадались, что каждый из методов поиска, как в глубину, так и в ширину, работает вслепую. Это методы поиска решения, которые полагаются исключительно на передвижение от одной вершины (цели) к другой без учета компьютером каких-либо обоснованных догадок. В некоторых ситуациях, когда объем области поиска контролируется, а также известно, что один метод лучше другого, такой подход может быть приемлемым. Однако для более универсальной программы искусственного интеллекта нужна процедура поиска, которая в среднем лучше любого из этих двух методов. Единственный способ получить такую процедуру — это добавить эвристические возможности.

Эвристика — это набор простых правил, которые позволяют оценить вероятность того, что поиск ведется в нужном направлении. Например, представьте, что вы заблудились в лесу и вас мучит жажда. В этом лесу деревья растут так густо, что далеко впереди ничего не видно, а сами деревья такие большие, что взобраться на них и осмотреться вокруг нельзя. Однако вам известно, что реки, ручьи и пруды вероятнее всего находятся в долинах; что животные часто протаптывают тропы к местам водопоя; что находясь поблизости от воды, ее можно чувствовать по “запаху”; а также вы знаете, что шум бегущей воды можно услышать. Таким образом, вы начинаете спускаться с холма вниз, ведь маловероятно, чтобы вода была на его вершине. Затем вы натываетесь на оленьи следы, которые также ведут вниз. Зная, что эти следы могут привести к воде, вы по ним и направляетесь. Через некоторое время с левой стороны начинает доноситься легкий шум. Предполагая, что это может быть вода, вы осторожно идете в этом направлении. По мере движения обнаруживается, что влажность воздуха усилилась; вода уже чувствуется по запаху. И, наконец, вы находите ручей и утоляете жажду. Как видите, эвристическая информация, не обязательно должна быть точной или гарантировать успех, однако она увеличивает шансы на то, что с помощью этого метода поиска вы найдете цель быстрее или найдете более оптимальное решение, а может быть, одновременно выполнятся оба эти условия. Короче говоря, применение эвристики увеличивает шансы скорейшего достижения успеха.

Вы, возможно, думаете, что эвристическую информацию можно легко ввести в программы, предназначенные для специального применения, но общие эвристические методы поиска создать невозможно. Это не так. Чаше всего эвристические методы поиска строятся на основе поиска максимума или минимума некоторого параметра решаемой задачи. И действительно, два эвристических подхода, с которыми мы познакомимся, используют противоположные эвристики и генерируют разные результаты. В основе этих двух методов поиска лежат процедуры поиска в глубину.

Поиск методом наискорейшего подъема

В задаче организации полета из Нью-Йорка в Лос-Анджелес могут быть два ограничивающих параметра, которые пассажиру хотелось бы свести к минимуму. Первый из них — это количество авиарейсов, необходимых, чтобы добраться до Лос-Анджелеса. А второй — это длина маршрута. Помните, что самый короткий маршрут — это не обязательно тот, который имеет минимальное количество авиарейсов¹. В алгоритме поиска, ищущем в качестве первого решения маршрут с минимальным количеством авиарейсов², применяется следующая эвристика. Чем длиннее авиарейс,

¹ Действительно, пересадок может быть мало (например, всего лишь одна), но иногда лучше совершить несколько коротких перелетов, чем два длинных. — *Прим. ред.*

² Т.е. маршрут с минимальным количеством пересадок. — *Прим. ред.*

то тем больше вероятность того, что путешественник окажется ближе к месту назначения. Поэтому количество авиарейсов¹ сводится к минимуму.

На языке искусственного интеллекта это называется *наискорейшим подъемом*. Алгоритм наискорейшего подъема в качестве следующей выбирает вершину, которая, как ему кажется, ближе всего находится к цели (то есть дальше всего от текущей вершины). Своим названием алгоритм обязан вот чему. Представьте себе, что турист-пешеход на полпути к вершине заблудился в темноте. И хотя вокруг темно, но турист, зная, что его лагерь находится на вершине, может, следуя алгоритму наискорейшего подъема, найти свой лагерь. Просто он должен помнить, что каждый шаг вверх — это шаг в правильном направлении.

Применительно к базе данных авиарейсов, в программе, генерирующей маршруты, эвристику наискорейшего подъема можно использовать следующим образом. Среди авиарейсов, исходящих из текущей вершины, выбирайте самый дальний — в надежде, что он доставит ближе всего к месту назначения. Вот как для этого нужно модифицировать подпрограмму `find()`:

```
/* Зная пункт отправления (параметр from), найти пункт прибытия
   для самого дальнего авиарейса (параметр anywhere). */
int find(char *from, char *anywhere)
{
    int pos, dist;

    pos=dist = 0;
    find_pos = 0;

    while(find_pos < f_pos) {
        if(!strcmp(flight[find_pos].from, from) &&
           !flight[find_pos].skip) {
            if(flight[find_pos].distance>dist) {
                pos = find_pos;
                dist = flight[find_pos].distance;
            }
        }
        find_pos++;
    }
    if(pos) {
        strcpy(anywhere, flight[pos].to);
        flight[pos].skip = 1;
        return flight[pos].distance;
    }
    return 0;
}
```

Теперь подпрограмма `find()` проводит поиск по всей базе данных, отыскивая самый дальний авиарейс из пункта вылета.

Вот вся программа, в которой используется алгоритм наискорейшего подъема:

```
/* Наискорейший подъем */
#include <stdio.h>
#include <string.h>

#define MAX 100

/* структура базы данных авиарейсов */
struct FL {
```

¹ Т.е. количество проходимых ребер графа. — Прим. ред.

```

    char from[20];
    char to[20];
    int distance;
    char skip; /* используется поиском с возвратом */
};

struct FL flight[MAX]; /* массив структур БД */

int f_pos = 0; /* количество записей в БД авиарейсов */
int find_pos = 0; /* индекс для поиска в БД авиарейсов */

int tos = 0; /* вершина стека */
struct stack {
    char from[20];
    char to[20];
    int dist;
};

struct stack bt_stack[MAX]; /* стек, используемый для поиска
с возвратом */

void setup(void), route(char *to);
void assert_flight(char *from, char *to, int dist);
void push(char *from, char *to, int dist);
void pop(char *from, char *to, int *dist);
void isflight(char *from, char *to);
int find(char *from, char *anywhere);
int match(char *from, char *to);

int main(void)
{
    char from[20], to[20];

    setup();

    printf("Пункт вылета: ");
    gets(from);
    printf("Пункт прибытия: ");
    gets(to);

    isflight(from, to);
    route(to);

    return 0;
}

/* Инициализация базы данных авиарейсов. */
void setup(void)
{
    assert_flight("Нью-Йорк", "Чикаго", 1000);
    assert_flight("Чикаго", "Денвер", 1000);
    assert_flight("Нью-Йорк", "Торонто", 800);
    assert_flight("Нью-Йорк", "Денвер", 1900);
    assert_flight("Торонто", "Калгари", 1500);
    assert_flight("Торонто", "Лос-Анджелес", 1800);
    assert_flight("Торонто", "Чикаго", 500);
    assert_flight("Денвер", "Урбана", 1000);
    assert_flight("Денвер", "Хьюстон", 1500);
}

```

```

    assert_flight("Хьюстон", "Лос-Анджелес", 1500);
    assert_flight("Денвер", "Лос-Анджелес", 1000);
}

/* Записать факты в базу данных. */
void assert_flight(char *from, char *to, int dist)
{
    if(f_pos < MAX) {
        strcpy(flight[f_pos].from, from);
        strcpy(flight[f_pos].to, to);
        flight[f_pos].distance = dist;
        flight[f_pos].skip = 0;
        f_pos++;
    }
    else printf("База данных авиарейсов заполнена.\n");
}

/* Показать маршрут и общее расстояние. */
void route(char *to)
{
    int dist, t;

    dist = 0;
    t = 0;
    while(t < tos) {
        printf("%s - ", bt_stack[t].from);
        dist += bt_stack[t].dist;
        t++;
    }
    printf("%s\n", to);
    printf("Расстояние в милях равно %d.\n", dist);
}

/* Если между двумя городами имеется авиарейс, то возвращается
   расстояние между ними, а в противном случае возвращается 0. */
int match(char *from, char *to)
{
    register int t;

    for(t=f_pos-1; t > -1; t--)
        if(!strcmp(flight[t].from, from) &&
            !strcmp(flight[t].to, to)) return flight[t].distance;

    return 0; /* не найден */
}

/* Зная пункт отправления (параметр from), найти пункт прибытия
   для самого дальнего авиарейса (параметр anywhere). */
int find(char *from, char *anywhere)
{
    int pos, dist;

    pos=dist = 0;
    find_pos = 0;

    while(find_pos < f_pos) {
        if(!strcmp(flight[find_pos].from, from) &&

```



```

        !flight[find_pos].skip) {
            if(flight[find_pos].distance>dist) {
                pos = find_pos;
                dist = flight[find_pos].distance;
            }
        }
        find_pos++;
    }
    if(pos) {
        strcpy(anywhere, flight[pos].to);
        flight[pos].skip = 1;
        return flight[pos].distance;
    }
    return 0;
}

/* Определить, имеется ли маршрут из города, на который указывает
   параметр from в город, на который указывает параметр to. */
void isflight(char *from, char *to)
{
    int d, dist;
    char anywhere[20];

    if(d=match(from, to)) {
        /* это цель */
        push(from, to, d);
        return;
    }

    /* найти любой авиарейс */
    if(dist=find(from, anywhere)) {
        push(from, to, dist);
        isflight(anywhere, to);
    }
    else if(tos > 0) {
        pop(from, to, &dist);
        isflight(from, to);
    }
}

/* Подпрограммы обращения к стеку */
void push(char *from, char *to, int dist)
{
    if(tos < MAX) {
        strcpy(bt_stack[tos].from, from);
        strcpy(bt_stack[tos].to, to);
        bt_stack[tos].dist = dist;
        tos++;
    }
    else printf("Стек заполнен.\n");
}

void pop(char *from, char *to, int *dist)
{
    if(tos > 0) {
        tos--;
        strcpy(from, bt_stack[tos].from);
        strcpy(to, bt_stack[tos].to);
    }
}

```

```

    *dist = bt_stack[tos].dist;
}
else printf("Стек пуст.\n");
}

```

В результате выполнения программы получается такое решение:

Нью-Йорк - Денвер - Лос-Анджелес
 Расстояние в милях равно 2900.

Оно довольно-таки хорошее! В полученном маршруте количество пересадок минимально (только одна), и он достаточно близок к самому короткому маршруту. Более того, программа приближается к решению, не тратя времени и усилий на обширные возвраты.

Однако если бы отсутствовал авиарейс между Денвером и Лос-Анджелесом, то решение не было бы таким хорошим. Это был бы маршрут Нью-Йорк — Денвер — Хьюстон — Лос-Анджелес общей протяженностью 4900 миль! При получении решения пришлось бы делать восхождение на ложный максимум. Как можно легко увидеть, перелет в Хьюстон не приблизит нас к цели, то есть к Лос-Анджелесу. На рис. 25.7 показано как первое решение, так и путь к ложному максимуму.

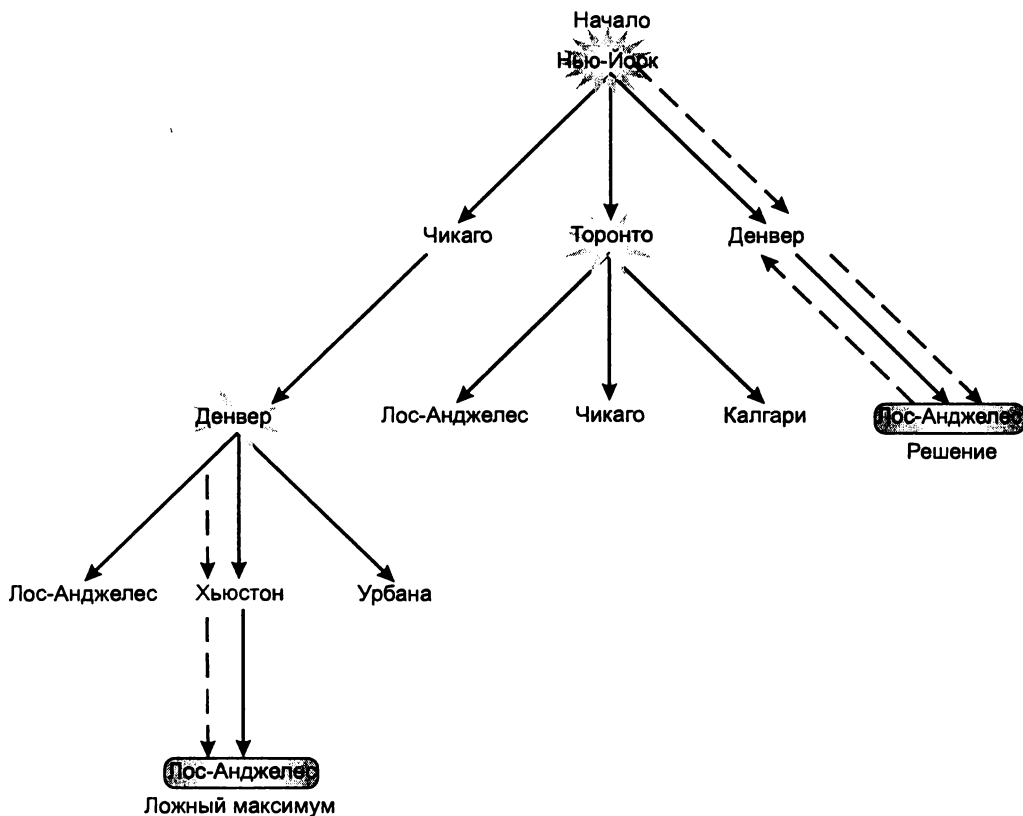


Рис. 25.7. Пути к решению и на ложный максимум, найденные с помощью наискорейшего подъема

Анализ наискорейшего подъема

Наискорейший подъем дает во многих случаях достаточно хорошие решения, потому что обычно перед тем как решение будет найдено, надо посещать сравнительно мало вершин. Однако у этого метода могут быть такие три недостатка. Во-первых, имеется проблема ложных максимумов, которой мы просто по счастливой случайности смогли избежать во втором решении нашего примера. Именно для того чтобы избежать ложных максимумов, при поиске решения приходится широко использовать возвраты. Вторая проблема связана с “плато” — ситуациями, когда все следующие шаги выглядят одинаково хорошими (или плохими). В таком случае наискорейший подъем будет не лучше, чем поиск в глубину. И последней проблемой является “горный хребет”. В таком случае наискорейший подъем проходит плохо, так как алгоритм при возвратах заставляет несколько раз пересекать “горный хребет”.

Но несмотря на возможные проблемы, наискорейший подъем, как правило, быстрее любого неэвристического метода приводит к решению, которое близко к оптимальному.

Поиск с использованием частичного пути минимальной стоимости

Противоположностью наискорейшему подъему является *поиск с использованием частичного пути минимальной стоимости*. Эта стратегия похожа на то, как если бы вы стояли на середине улицы, ведущей на большую гору, а на ногах у вас были бы надежные роликовые коньки. Вы бы тогда явно почувствовали, что двигаться вниз намного легче, чем вверх! Другими словами, поиск с использованием частичного пути минимальной стоимости выбирает путь наименьшего сопротивления.

Если поиск с использованием частичного пути минимальной стоимости применять к задаче выбора маршрутов полетов, то это означает, что авиарейсы всегда выбираются самые короткие — в надежде, что и найденный маршрут окажется самым коротким. В отличие от наискорейшего подъема, который стремится уменьшить количество авиарейсов, поиск с использованием частичного пути минимальной стоимости сводит к минимуму общую длину маршрута.

Чтобы использовать поиск с использованием частичного пути минимальной стоимости, сначала, как обычно, нужно переписать функцию `find()`. Ниже показан ее новый код.

```
/* Найти самый близкий город и поместить его в "anywhere". */
int find(char *from, char *anywhere)
{
    int pos, dist;

    pos = 0;
    dist = 32000; /* больше длины самого длинного авиарейса */
    find_pos = 0;

    while(find_pos < f_pos) {
        if(!strcmp(flight[find_pos].from, from) &&
            !flight[find_pos].skip) {
            if(flight[find_pos].distance < dist) {
                pos = find_pos;
                dist = flight[find_pos].distance;
            }
        }
        find_pos++;
    }
```

```

    }
    if(pos) {
        strcpy( anywhere, flight[pos].to);
        flight[pos].skip = 1;
        return flight[pos].distance;
    }
    return 0;
}

```

С помощью этой версии find() получается такое решение:

Нью-Йорк - Торонто - Лос-Анджелес
 Расстояние в милях равно 2600.

Как видите, в данном случае этот метод поиска позволяет найти самый короткий маршрут. Цепочка, ведущая к цели (т.е. маршрут, причем самый короткий), показана на рис. 25.8.

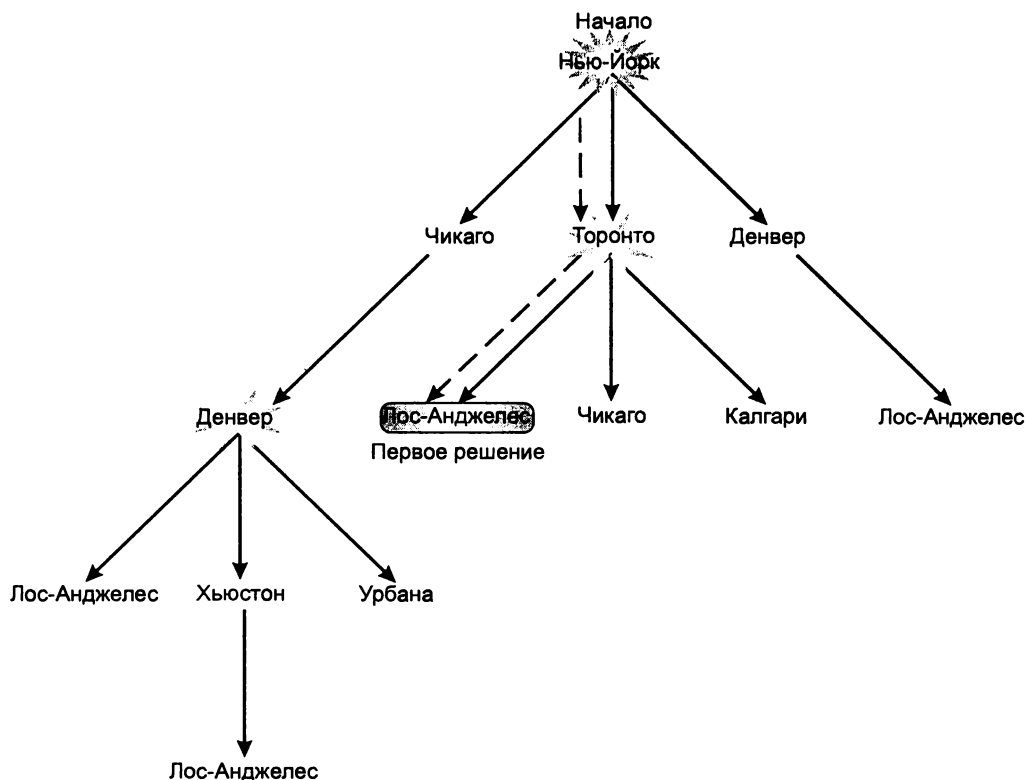


Рис. 25.8. Эта цепочка, ведущая к решению (т.е. путь, притом наикратчайший), была найдена методом поиска с использованием частичного пути минимальной стоимости

Анализ поиска с использованием частичного пути минимальной стоимости

Поиск с использованием частичного пути минимальной стоимости и наискорейший подъем имеют одни и те же достоинства и недостатки, но только с точностью до наоборот: то, что является достоинством одного метода, является недостатком другого,

и наоборот. При поиске с использованием частичного пути минимальной стоимости могут появиться ложные, обманчивые “овраги, долины, низины и пропасти”, но в целом этот метод работает достаточно хорошо. Однако не надо думать, что если поиск с использованием частичного пути минимальной стоимости работал в нашей задаче лучше, чем наискорейший подъем, то он вообще работает лучше¹. Т.е. можно сказать, что в среднем он работает лучше, чем поиск вслепую.

Выбор метода поиска

Как вы видели, обычно эвристические методы работают в среднем лучше, чем методы поиска вслепую. Однако не всегда представляется возможным использовать эвристический поиск. Так происходит потому, что иногда недостаточно информации для определения вероятности того, что следующая вершина ведет к цели. Поэтому правила, определяющие выбор метода поиска, делятся на две категории: одни применяются для задач, в которых можно использовать эвристический поиск, другие — для задач, в которых эвристический поиск применить нельзя².

Если для задачи не удастся найти достаточно эффективную эвристику, то лучшим методом обычно является поиск в глубину. Единственным исключением может быть тот случай, когда вам известно нечто такое, что говорит в пользу поиска в ширину.

Выбор между наискорейшим подъемом и поиском с использованием частичного пути минимальной стоимости на самом деле состоит в том, что вы решаете, какой именно параметр требуется минимизировать или максимизировать. Вообще-то, в среднем наискорейший подъем генерирует решение, имеющее наименьшее количество вершин, а поиск с использованием частичного пути минимальной стоимости находит путь наименьшей длины.

Допустим, вы ищете решение, близкое к оптимальному, однако по каким-либо причинам нельзя использовать исчерпывающий поиск. Тогда можно воспользоваться следующим эффективным методом: применить все четыре метода поиска, а затем выбрать наилучшее решение. В сущности, все методы поиска работают по-разному, поэтому у одного из них результат должен все-таки быть лучше, чем у остальных.

Поиск нескольких решений

Иногда бывает полезно найти несколько решений одной задачи. Но это не то же самое, что полный перебор, при котором необходимо найти все решения. Например, подумайте о проектировании дома вашей мечты. Вам, чтобы решить, какой проект будет наилучшим, может потребоваться несколько разных набросков поэтажного плана дома, но наброски *всех возможных домов* вам не нужны. В сущности, разные решения помогают увидеть много разных подходов к поиску окончательного решения, а затем реализовать один из них.

Генерировать разные решения можно несколькими способами, но здесь рассказывается только о двух из них. Первый — это удаление путей, а второй — удаление вершин. Как и следует из названий этих методов, чтобы при генерации разных решений

¹ Действительно, зная условие задачи, в которой поиск с использованием частичного пути минимальной стоимости работает лучше, можно сформулировать двойственную задачу, в которой будет лучше работать наискорейший подъем. — *Прим. ред.*

² Обратите внимание, это утверждение означает, что общие эвристики могут оказаться неэффективными. Более того, для каждой, пусть даже самой “интеллектуальной” эвристики, найдется (непустой) класс задач, для которых “алгоритм Британского музея” (полный перебор без каких-либо правил предпочтения) будет более эффективным. — *Прим. ред.*

те из них, которые уже найдены не повторялись, некоторые элементы требуется удалять из системы. Помните, что ни один из этих методов не является попыткой найти все решения (и даже не может для этого использоваться). Поиск всех решений — это совсем другая задача, за выполнение которой обычно даже не берутся, потому что она подразумевает исчерпывающий поиск.

Удаление путей

При использовании метода генерации нескольких решений, который называется методом *удаления путей*, из базы данных удаляются все вершины текущего решения, а затем делается попытка найти следующее решение. В сущности, при удалении путей подрезаются ветви дерева.

Чтобы найти несколько решений с помощью удаления путей, необходимо в программе поиска в глубину изменить функцию `main()`:

```
int main(void)
{
    char from[20], to[20];

    setup();

    printf("Пункт вылета: ");
    gets(from);
    printf("Пункт прибытия: ");
    gets(to);
    do {
        isflight(from, to);
        route(to);
        tos = 0; /* сброс стека, используемого при поиске с возвратом */
    } while(getchar() != 'q'); /* для выхода вводится символ 'q' */

    return 0;
}
```

У каждого авиарейса, входящего в цепочку, ведущую к решению, будет помечено его поле `skip`. Следовательно, такие авиарейсы больше не будут найдены функцией `find()`, все авиарейсы, имеющиеся в решении, будут эффективно удалены. Перед поиском следующего решения нужно только сбрасывать переменную `tos`, ведь именно это на самом деле и очищает стек, используемый при поиске с возвратом.

Метод удаления путей обнаруживает следующие решения:

```
Нью-Йорк - Чикаго - Денвер - Лос-Анджелес
Расстояние в милях равно 3000.
Нью-Йорк - Торонто - Лос-Анджелес
Расстояние в милях равно 2600.
Нью-Йорк - Денвер - Лос-Анджелес
Расстояние в милях равно 2900.
```

При поиске было найдено три наилучших решения. Однако этот результат нельзя обобщать, так как он зависит от размещения информации в базе данных и от конкретной ситуации.

Удаление вершин

Для генерации нескольких решений также применяется метод *удаления вершин*. Используя этот метод, из пути, представляющего собой найденное решение, удаляется последняя вершина, а затем делается повторная попытка найти решение. Для этого

функция `main()` с помощью другой функции `retract()` должна выталкивать из стека, используемого для поиска с возвратом, последнюю вершину и удалять ее из базы данных. Кроме того, все поля `skip` должны сбрасываться с помощью функции `clearmarkers()`. Необходимо также очищать стек, используемый для поиска с возвратом. Ниже приведены коды функций `main()`, `clearmarkers()` и `retract()`:

```
int main(void)
{
    char from[20], to[20], c1[20], c2[20];
    int d;

    setup();

    printf("Пункт вылета: ");
    gets(from);
    printf("Пункт прибытия: ");
    gets(to);
    do {
        isflight(from,to);
        route(to);
        clearmarkers(); /* переустановка базы данных */
        if(tos > 0) pop(c1,c2,&d);
        retract(c1,c2); /* удаление последней вершины из базы данных */
        tos = 0; /* сброс стека, используемого для поиска с возвратом */
    } while(getchar() != 'q'); /* для выхода вводится символ 'q' */

    return 0;
}

/* Сбросить поле skip, т.е. заново активизировать все вершины. */
void clearmarkers()
{
    int t;

    for(t=0; t < f_pos; ++t) flight[t].skip = 0;
}

/* Удаление записи из базы данных. */
void retract(char *from, char *to)
{
    int t;

    for(t=0; t < f_pos; t++)
        if(!strcmp(flight[t].from, from) &&
            !strcmp(flight[t].to, to)) {
            strcpy(flight[t].from, "");
            return;
        }
}
```

Как видите, запись удаляется таким образом: имя города заменяется строкой нулевой длины. Ниже приведен полный текст программы, в которой используется метод удаления вершин:

```
/* Поиск нескольких решений методом поиска в глубину;
   некоторые вершины удаляются */
#include <stdio.h>
#include <string.h>
```

```

#define MAX 100

/* структура базы данных авиарейсов */
struct FL {
    char from[20];
    char to[20];
    int distance;
    char skip; /* используется для поиска с возвратом */
};

struct FL flight[MAX];

int f_pos = 0; /* количество записей в БД авиарейсов */
int find_pos = 0; /* индекс для поиска в БД авиарейсов */

int tos = 0; /* вершина стека */
struct stack {
    char from[20];
    char to[20];
    int dist;
};
struct stack bt_stack[MAX]; /* стек, используемый для поиска
                             с возвратом */

void retract(char *from, char *to);
void clearmarkers(void);
void setup(void), route(char *to);
void assert_flight(char *from, char *to, int dist);
void push(char *from, char *to, int dist);
void pop(char *from, char *to, int *dist);
void isflight(char *from, char *to);
int find(char *from, char *anywhere);
int match(char *from, char *to);

int main(void)
{
    char from[20], to[20], c1[20], c2[20];
    int d;

    setup();

    printf("Пункт вылета: ");
    gets(from);
    printf("Пункт прибытия: ");
    gets(to);
    do {
        isflight(from, to);
        route(to);
        clearmarkers(); /* возврат базы данных в исходное состояние */
        if(tos > 0) pop(c1, c2, &d);
        retract(c1, c2); /* удаление последней вершины из базы данных */
        tos = 0; /* сброс стека, используемого для поиска с возвратом */
    } while(getchar() != 'q'); /* для выхода вводится символ 'q' */

    return 0;
}

```



```

/* Инициализация базы данных авиарейсов. */
void setup(void)
{
    assert_flight("Нью-Йорк", "Чикаго", 1000);
    assert_flight("Чикаго", "Денвер", 1000);
    assert_flight("Нью-Йорк", "Торонто", 800);
    assert_flight("Нью-Йорк", "Денвер", 1900);
    assert_flight("Торонто", "Калгари", 1500);
    assert_flight("Торонто", "Лос-Анджелес", 1800);
    assert_flight("Торонто", "Чикаго", 500);
    assert_flight("Денвер", "Урбана", 1000);
    assert_flight("Денвер", "Хьюстон", 1500);
    assert_flight("Хьюстон", "Лос-Анджелес", 1500);
    assert_flight("Денвер", "Лос-Анджелес", 1000);
}

/* Запомнить авиарейсы в базе данных. */
void assert_flight(char *from, char *to, int dist)
{
    if(f_pos < MAX) {
        strcpy(flight[f_pos].from, from);
        strcpy(flight[f_pos].to, to);
        flight[f_pos].distance = dist;
        flight[f_pos].skip = 0;
        f_pos++;
    }
    else printf("База данных авиарейсов заполнена.\n");
}

/* Сбросить поле skip, то есть заново активизировать все вершины. */
void clearmarkers()
{
    int t;

    for(t=0; t < f_pos; ++t) flight[t].skip = 0;
}

/* Удаление записи из базы данных. */
void retract(char *from, char *to)
{
    int t;

    for(t=0; t < f_pos; t++)
        if(!strcmp(flight[t].from, from) &&
            !strcmp(flight[t].to, to)) {
            strcpy(flight[t].from, "");
            return;
        }
}

/* Показать маршрут и общее расстояние. */
void route(char *to)
{
    int dist, t;

    dist = 0;
    t = 0;
    while(t < tos) {

```

```

        printf("%s - ", bt_stack[t].from);
        dist += bt_stack[t].dist;
        t++;
    }
    printf("%s\n", to);
    printf("Расстояние в милях равно %d.\n", dist);
}

/* Зная пункт отправления (параметр from), найти пункт прибытия
(параметр anywhere). */
int find(char *from, char *anywhere)
{
    find_pos = 0;
    while(find_pos < f_pos) {
        if(!strcmp(flight[find_pos].from, from) &&
            !flight[find_pos].skip) {
            strcpy(anywhere, flight[find_pos].to);
            flight[find_pos].skip = 1;
            return flight[find_pos].distance;
        }
        find_pos++;
    }
    return 0;
}

/* Если между двумя городами (параметры from и to) имеется авиарейс,
то возвращается расстояние между ними, а в противном случае
возвращается 0. */
int match(char *from, char *to)
{
    register int t;

    for(t=f_pos-1; t > -1; t--)
        if(!strcmp(flight[t].from, from) &&
            !strcmp(flight[t].to, to)) return flight[t].distance;

    return 0; /* не найден */
}

/* Определить, имеется ли маршрут между двумя городами, на которые
указывают параметры from (из) и to (в). */
void isflight(char *from, char *to)
{
    int d, dist;
    char anywhere[20];

    if(d=match(from, to)) {
        push(from, to, d); /* расстояние */
        return;
    }

    if(dist=find(from, anywhere)) {
        push(from, to, dist);
        isflight(anywhere, to);
    }
    else if(tos > 0) {
        pop(from, to, &dist);
        isflight(from, to);
    }
}

```

```

    }
}

/* Подпрограммы обращения к стеку */
void push(char *from, char *to, int dist)
{
    if(tos < MAX) {
        strcpy(bt_stack[tos].from, from);
        strcpy(bt_stack[tos].to, to);
        bt_stack[tos].dist = dist;
        tos++;
    }
    else printf("Стек заполнен.\n");
}

void pop(char *from, char *to, int *dist)
{
    if(tos > 0) {
        tos--;
        strcpy(from, bt_stack[tos].from);
        strcpy(to, bt_stack[tos].to);
        *dist = bt_stack[tos].dist;
    }
    else printf("Стек пуст.\n");
}

```

С помощью этого метода получаются такие решения:

```

Нью-Йорк - Чикаго - Денвер - Лос-Анджелес
Расстояние в милях равно 3000.
Нью-Йорк - Чикаго - Денвер - Хьюстон - Лос-Анджелес
Расстояние в милях равно 5000.
Нью-Йорк - Торонто - Лос-Анджелес
Расстояние в милях равно 2600.

```

В этом случае второе решение — это самый худший из возможных маршрутов, но оптимальное решение все равно найдено. Однако помните, что эти результаты нельзя обобщать: они зависят как от физической организации базы данных, так и от конкретной ситуации.



Поиск “оптимального” решения

Все предыдущие методы поиска от начала и до конца предназначены для нахождения решения, притом любого из них. Кроме того, те из приведенных методов, которые являются эвристическими, показали, что при определенных стараниях решение можно до некоторой степени улучшить. Но не было сделано ни одной попытки убедиться, что получено именно оптимальное решение. Но иногда бывает так, что подходит *только* оптимальное решение. Впрочем, не забывайте, что “оптимальное решение” в данном смысле просто означает наилучший маршрут, который можно найти с помощью одного из разнообразных методов, генерирующих варианты решений. И этот маршрут в действительности может и не быть наилучшим решением. (Ведь для получения действительно оптимального решения может понадобиться полный перебор, на который требуется недопустимо много времени.)

Перед тем как закончить изучение хорошо проработанного примера с маршрутами, проанализируйте программу, которая находит оптимальный маршрут при следующем ограничении: общее расстояние должно быть минимальным. Эта программа с помо-

стью удаления путей генерирует разные решения, а с помощью поиска с использованием частичного пути минимальной стоимости получает путь с минимальным общим расстоянием. При нахождении самого короткого пути применяют такой принцип: из двух решений, старого и нового, сохраняется лишь то, которое короче. А когда новые решения больше не генерируются, то остается только оптимальное.

Чтобы реализовать этот алгоритм, необходимо серьезно изменить функцию `route()` и создать еще один стек. В новом стеке будет храниться текущее решение, а в конце работы — оптимальное. Этот стек называется `solution` (решение), а вот и сама измененная функция `route()`:

```
/* Найти кратчайшее расстояние. */
int route(void)
{
    int dist, t;
    static int old_dist=32000;

    if(!tos) return 0; /* все сделано */
    t = 0;
    dist = 0;
    while(t < tos) {
        dist += bt_stack[t].dist;
        t++;
    }

    /* Если короче, то найти новое решение */
    if(dist < old_dist && dist) {
        t = 0;
        old_dist = dist;
        stos = 0; /* удалить старый маршрут из стека solution */
        while(t < tos) {
            spush(bt_stack[t].from, bt_stack[t].to, bt_stack[t].dist);
            t++;
        }
    }
    return dist;
}
```

Далее приводится вся программа. Обратите внимание на изменения в функции `main()` и на то, что в программу введена функция `spush()`, которая помещает вершины нового решения в стек решений (`solution`).

```
/* Поиск оптимального решения методом поиска частичного пути
   минимальной стоимости;
   некоторые маршруты удаляются.
*/
#include <stdio.h>
#include <string.h>

#define MAX 100

/* структура базы данных авиарейсов */
struct FL {
    char from[20];
    char to[20];
    int distance;
    char skip; /* используется при поиске с возвратом */
};
```

```

struct FL flight[MAX]; /* массив структур БД */

int f_pos = 0; /* количество записей в БД авиарейсов */
int find_pos = 0; /* индекс для поиска в БД авиарейсов */

int tos = 0; /* вершина стека */
int stos = 0; /* вершина стека solution */

struct stack {
    char from[20];
    char to[20];
    int dist;
} ;

struct stack bt_stack[MAX]; /* стек, используемый для поиска
с возвратом */
struct stack solution[MAX]; /* хранит временные решения */

void setup(void);
int route(void);
void assert_flight(char *from, char *to, int dist);
void push(char *from, char *to, int dist);
void pop(char *from, char *to, int *dist);
void isflight(char *from, char *to);
void spush(char *from, char *to, int dist);
int find(char *from, char *anywhere);
int match(char *from, char *to);

int main(void)
{
    char from[20], to[20];
    int t, d;

    setup();

    printf("Пункт вылета: ");
    gets(from);
    printf("Пункт прибытия: ");
    gets(to);
    do {
        isflight(from, to);
        d = route();
        tos = 0; /* возврат в исходное состояние стека, используемого
для поиска с возвратом */
    } while(d != 0); /* пока алгоритм может найти новые решения */

    t = 0;
    printf("Оптимальное решение:\n");
    while(t < stos) {
        printf("%s - ", solution[t].from);
        d += solution[t].dist;
        t++;
    }
    printf("%s\n", to);
    printf("Расстояние в милях равно %d.\n", d);

    return 0;
}

```

```

/* Инициализация базы данных авиарейсов. */
void setup(void)
{
    assert_flight("Нью-Йорк", "Чикаго", 1000);
    assert_flight("Чикаго", "Денвер", 1000);
    assert_flight("Нью-Йорк", "Торонто", 800);
    assert_flight("Нью-Йорк", "Денвер", 1900);
    assert_flight("Торонто", "Калгари", 1500);
    assert_flight("Торонто", "Лос-Анджелес", 1800);
    assert_flight("Торонто", "Чикаго", 500);
    assert_flight("Денвер", "Урбана", 1000);
    assert_flight("Денвер", "Хьюстон", 1500);
    assert_flight("Хьюстон", "Лос-Анджелес", 1500);
    assert_flight("Денвер", "Лос-Анджелес", 1000);
}

/* Записать факты в базу данных. */
void assert_flight(char *from, char *to, int dist)
{
    if(f_pos < MAX) {
        strcpy(flight[f_pos].from, from);
        strcpy(flight[f_pos].to, to);
        flight[f_pos].distance = dist;
        flight[f_pos].skip = 0;
        f_pos++;
    }
    else printf("База данных авиарейсов заполнена.\n");
}

/* Найти кратчайшее расстояние. */
int route(void)
{
    int dist, t;
    static int old_dist=32000;

    if(!tos) return 0; /* все сделано */
    t = 0;
    dist = 0;
    while(t < tos) {
        dist += bt_stack[t].dist;
        t++;
    }

    /* Если короче, то заменить новым решением */
    if(dist < old_dist && dist) {
        t = 0;
        old_dist = dist;
        stos = 0; /* удалить старый маршрут из стека solution */
        while(t < tos) {
            spush(bt_stack[t].from, bt_stack[t].to, bt_stack[t].dist);
            t++;
        }
    }
    return dist;
}

/* Если между двумя городами (параметры from и to) имеется авиарейс,

```

```

    то возвращается расстояние между ними, в противном случае
    возвращается 0. */
int match(char *from, char *to)
{
    register int t;

    for(t=f_pos-1; t > -1; t--)
        if(!strcmp(flight[t].from, from) &&
            !strcmp(flight[t].to, to)) return flight[t].distance;

    return 0; /* не найден */
}

/* Зная пункт отправления (параметр from), найти пункт прибытия
(параметр anywhere). */
int find(char *from, char *anywhere)
{
    find_pos = 0;
    while(find_pos < f_pos) {
        if(!strcmp(flight[find_pos].from, from) &&
            !flight[find_pos].skip) {
            strcpy(anywhere, flight[find_pos].to);
            flight[find_pos].skip = 1;
            return flight[find_pos].distance;
        }
        find_pos++;
    }
    return 0;
}

/* Определить, имеется ли маршрут между двумя городами, на которые
указывают параметры from (из) и to (в). */
void isflight(char *from, char *to)
{
    int d, dist;
    char anywhere[20];

    if(d=match(from, to)) {
        push(from, to, d); /* расстояние */
        return;
    }

    if(dist=find(from, anywhere)) {
        push(from, to, dist);
        isflight(anywhere, to);
    }
    else if(tos > 0) {
        pop(from, to, &dist);
        isflight(from, to);
    }
}

/* Подпрограммы обращения к стеку */
void push(char *from, char *to, int dist)
{
    if(tos < MAX) {
        strcpy(bt_stack[tos].from, from);

```

```

        strcpy(bt_stack[tos].to, to);
        bt_stack[tos].dist = dist;
        tos++;
    }
    else printf("Стек заполнен.\n");
}

void pop(char *from, char *to, int *dist)
{
    if(tos > 0) {
        tos--;
        strcpy(from, bt_stack[tos].from);
        strcpy(to, bt_stack[tos].to);
        *dist = bt_stack[tos].dist;
    }
    else printf("Стек пуст.\n");
}

/* Стек решений (solution) */
void spush(char *from, char *to, int dist)
{
    if(stos < MAX) {
        strcpy(solution[stos].from, from);
        strcpy(solution[stos].to, to);
        solution[stos].dist = dist;
        stos++;
    }
    else printf("Стек для кратчайших маршрутов заполнен.\n");
}

```

В последнем методе есть один недостаток: обход по всем маршрутам продолжается вплоть до их листьев. Будь этот метод более совершенным, обход по маршруту немедленно прекращался бы тогда, когда длина найденной части маршрута достигала бы текущего минимума (или превосходила его). С учетом этого обстоятельства данную программу можно значительно улучшить.



И снова возвращаемся к поиску потерянных ключей

Возможно, вы помните, что вам все-таки удалось найти ключи от машины. (Если, конечно, не потеряли их снова.) Однако главу, посвященную решению задач, хотелось бы завершить демонстрацией программы, которая находит потерянные ключи от машины. Ведь подобные задачи встречаются так часто! В коде программы используются те же методы, что и при поиске маршрута из одного города в другой. Теперь, когда вы уже освоили технику применения языка С для решения задач, программа представлена без последующих подробных объяснений.

```

/* Найти ключи с помощью поиска в глубину. */
#include <stdio.h>
#include <string.h>

#define MAX 100

/* структура базы данных keys (ключи) */
struct FL {

```



```

    char from[20];
    char to[20];
    char skip;
};

struct FL keys[MAX]; /* массив структур БД */

int f_pos = 0; /* количество комнат в доме */
int find_pos = 0; /* индекс для поиска в БД keys */

int tos = 0; /* вершина стека */
struct stack {
    char from[20];
    char to[20];
} ;
struct stack bt_stack[MAX]; /* стек, используемый для поиска
                             с возвратом */

void setup(void), route(void);
void assert_keys(char *from, char *to);
void push(char *from, char *to);
void pop(char *from, char *to);
void iskeys(char *from, char *to);
int find(char *from, char *anywhere);
int match(char *from, char *to);

int main(void)
{
    char from[20] = "входная_дверь";
    char to[20] = "ключи";

    setup();
    iskeys(from, to);
    route();

    return 0;
}

/* Инициализация базы данных. */
void setup(void)
{
    assert_keys("входная_дверь", "гостиная");
    assert_keys("гостиная", "ванная");
    assert_keys("гостиная", "холл");
    assert_keys("холл", "спальня1");
    assert_keys("холл", "спальня2");
    assert_keys("холл", "большая_спальня");
    assert_keys("гостиная", "кухня");
    assert_keys("кухня", "ключи");
}

/* Запомнить факты в базе данных. */
void assert_keys(char *from, char *to)
{
    if(f_pos < MAX) {
        strcpy(keys[f_pos].from, from);
        strcpy(keys[f_pos].to, to);
        keys[f_pos].skip = 0;
    }
}

```

```

        f_pos++;
    }
    else printf("База данных keys заполнена.\n");
}

/* Показать путь к ключам. */
void route(void)
{
    int t;

    t = 0;
    while(t < tos) {
        printf("%s", bt_stack[t].from);
        t++;
        if(t < tos) printf(" - ");
    }
    printf("\n");
}

/* Посмотреть, есть ли ребро в графе. */
int match(char *from, char *to)
{
    register int t;

    for(t=f_pos-1; t > -1; t--)
        if(!strcmp(keys[t].from, from) &&
            !strcmp(keys[t].to, to)) return 1;

    return 0; /* не найдено */
}

/* Зная откуда (from), попасть куда-будь (anywhere). */
int find(char *from, char *anywhere)
{
    find_pos = 0;

    while(find_pos < f_pos) {
        if(!strcmp(keys[find_pos].from, from) &&
            !keys[find_pos].skip) {
            strcpy(anywhere, keys[find_pos].to);
            keys[find_pos].skip = 1;
            return 1;
        }
        find_pos++;
    }
    return 0;
}

/* Определить, имеется ли путь из from (из) в to (в). */
void iskeys(char *from, char *to)
{
    char anywhere[20];

    if(match(from, to)) {
        push(from, to); /* расстояние */
        return;
    }
}

```

```

    if(find(from, anywhere)) {
        push(from, to);
        iskeys(anywhere, to);
    }
    else if(tos > 0) {
        pop(from, to);
        iskeys(from, to);
    }
}

/* Подпрограммы обращения к стеку */
void push(char *from, char *to)
{
    if(tos < MAX) {
        strcpy(bt_stack[tos].from, from);
        strcpy(bt_stack[tos].to, to);
        tos++;
    }
    else printf("Стек заполнен.\n");
}

void pop(char *from, char *to)
{
    if(tos > 0) {
        tos--;
        strcpy(from, bt_stack[tos].from);
        strcpy(to, bt_stack[tos].to);
    }
    else printf("Стек пуст.\n");
}

```

Полный справочник по



Часть V

Разработка программ с помощью C

В этой части книги исследуются различные аспекты процесса разработки программ, обусловленные средой программирования на C. В главе 26 демонстрируются приемы использования C при создании скелета приложения, предназначенного для работы в среде Windows 2000. Глава 27 освещает процесс проектирования прикладных программ с помощью C. Глава 28 посвящена вопросам переноса, эффективности и отладки.

Полный
справочник по



Глава 26

**Создание скелета приложения
для Windows 2000**

Язык С как один из основных языков программирования имеет первостепенное значение при создании программ для платформы Windows. По существу, вполне логично было бы просто поместить в этой части книги примеры программирования для Windows. Однако Windows является очень большой и сложной средой программирования, поэтому, к сожалению, невозможно описать в одной главе все детали и тонкости, которые необходимо учитывать при написании Windows-приложений. Но с другой стороны, вполне *возможно* представить базовые элементы, которые являются общими для всех приложений. Затем эти элементы могут быть объединены в минимальный “скелет” (заготовку) прикладной Windows-программы, который сформирует фундамент для ваших собственных Windows-приложений.

С момента своего появления операционная система Windows претерпела несколько превращений. В период подготовки данной книги текущей версией была, например, Windows 2000. Поэтому материалы данной главы максимально адаптированы именно для этой версии Windows. Тем не менее, даже если вы работаете с более свежей или, наоборот, со старой версией Windows, большая часть обсуждаемых вопросов останутся по-прежнему актуальными.

На заметку

Данная глава является адаптацией материала, почерпнутого из моей книги Windows 2000 Programming from the Ground up — Berkeley, CA: Osborne/McGraw-Hill, 2000. Для тех, кто захочет глубже изучить секреты программирования для Windows 2000, эта книга окажется исключительно полезным руководством.

Общая картина специфики программирования для Windows 2000

Краеугольным камнем операционной системы Windows 2000 (как впрочем и любой из версий Windows в целом) является идея предоставления любому человеку, имеющему базовые навыки работы с данной системой, возможность сесть за компьютер и запустить практически любое приложение без специального предварительного обучения. Чтобы реализовать этот замысел, Windows предоставляет пользователю единообразный интерфейс. Теоретически, если вы можете работать в какой-либо одной Windows-ориентированной программе, то вы сможете работать и во всех остальных. Без сомнения, в реальной жизни для максимально эффективного использования возможностей наиболее полезных программ по-прежнему требуется определенная подготовка. Но, по крайней мере, это обучение может быть сведено до разъяснения того, *что делает* данная программа, а не того, *как* пользователь должен *взаимодействовать* с ней. Поэтому неудивительно, что значительная часть кода в Windows-приложениях предназначена для реализации интерфейса пользователя.

Необходимо иметь в виду (и это очень важный момент), что не всякая программа, выполняемая под управлением Windows 2000, автоматически предоставляет пользователю интерфейс, выполненный в Windows-стиле. Операционная система Windows определяет среду, которая только поддерживает и стимулирует единообразие, но не заставляет делать это в “принудительном порядке”. Например, можно написать массу Windows-программ, которые не будут использовать преимущества, которые обеспечивают стандартные элементы интерфейса Windows. (Кстати, очень многие системные и сетевые утилиты Windows выполнены именно в виде приложений командной строки, а не в стиле Windows. Но это допустимо, так как они предназначены для высококвалифицированных специалистов, например, системных администраторов.) Для того чтобы создать программу в стиле Windows, необходимо осознанно следовать определенным правилам. Только те программы, которые написаны с учетом использования

богатых возможностей Windows, будут выглядеть и вести себя как настоящие Windows-приложения. Конечно, вы можете отвергнуть базовую философию Windows-дизайна, но для такого решения у вас должны быть очень веские причины. Потому что иначе ваши программы будут нарушать самую фундаментальную заповедь Windows: обеспечить единообразный и последовательный интерфейс пользователя. В общем случае, если вы пишете прикладные программы под Windows 2000, то они обязаны соответствовать нормам и подходам проектирования стандартного Windows-стиля.

Давайте рассмотрим самые значительные составляющие, которые определяют среду приложений для Windows 2000.

Модель рабочего стола

Основная идея (с некоторыми нюансами) основанного на окнах пользовательского интерфейса состоит в том, чтобы обеспечить на экране монитора эквивалент поверхности рабочего стола. Так, на обычном письменном столе могут лежать стопкой несколько листов бумаги, причем часто некоторые листы торчат из стопки в разные стороны, так что под верхним листом видны фрагменты этих страниц. Эквивалентом рабочего стола в Windows 2000 является экран монитора. Эквивалентами листов бумаги или их фрагментов являются окна на экране. На обычном рабочем столе эти листы бумаги вы можете передвигать с места на место. Возможно, даже ухитряетесь выдернуть из стопки необходимый документ и положить его поверх всех остальных, или изучаете кусочек другого листа, выглядывающего из пачки документов. Аналогичные операции позволяет выполнять в своих окнах и Windows 2000. Благодаря возможности выбрать какое-то окно, вы можете сделать его активным. Это означает, что оно будет помещено поверх всех остальных окон, и вся информация, вводимая, например, с помощью клавиатуры будет направляться именно приложению, представленному этим окном. Окно можно увеличить или уменьшить, а также переместить в другое место экрана. Иными словами, Windows позволяет использовать поверхность экрана практически таким же образом, как и поверхность своего письменного стола. Все программы, соответствующие стандартам Windows-стиля, должны позволять пользователям выполнять подобные действия.

Мышь

Как и все предыдущие версии Windows, операционная система Windows 2000 использует манипулятор типа мышь почти во всех операциях управления, выбора и перемещения. Конечно, для этого можно использовать также и клавиатуру, но в первую очередь Windows оптимизирована для работы с мышкой. Поэтому ваши программы должны поддерживать мышь в качестве основного устройства ввода, причем везде, где это только возможно. К счастью, для основного набора обычных действий, как, например, использования линейки прокрутки, выбора элемента меню и тому подобного, автоматически предусмотрено использование мыши.

Пиктограммы, растровые изображения и другая графика

Windows 2000 широко поддерживает, даже поощряет применение пиктограмм, растровых изображений и других видов графики. Обоснованием повсеместного применения таких элементов служит старая проверенная временем поговорка: Лучше один раз увидеть, чем сто раз услышать¹. Пиктограмма — это маленький мнемонический значок, символизирующий некоторую операцию, ресурс или программу. Растровое изображение, часто называемое *битмапкой* (bitmap) — это графическое изображение прямоугольной формы в растровом формате; такие изображения часто используются

¹ Вот ее Windows-переформулировка: Лучше один раз шелкнуть на пиктограмме, чем сто раз набрать на клавиатуре. — *Прим. ред.*

для того, чтобы быстро донести до пользователя некоторую информацию в визуальном виде. Более того, растровые изображения могут также использоваться в качестве элементов меню. Windows 2000 поддерживает очень широкий диапазон графических возможностей, в том числе прорисовку линий, прямоугольников и окружностей. Правильное применение подобных графических элементов является необходимым условием успешного программирования для Windows.

Меню, средства управления и диалоговые окна

Windows обеспечивает несколько видов стандартных элементов, которые предназначены для ввода информации пользователем. В их число входят: меню, разнообразные средства управления, а также диалоговые окна. Не отвлекаясь на подробности, можно сказать, что меню отображает варианты действий, на которых пользователь может остановить свой выбор. Поскольку меню являются стандартными элементами Windows-программирования, функции встроенного в меню выбора обеспечиваются средствами самой системы Windows. Из этого следует, что вашей программе не понадобится самой нести бремя непроизводительных организационных издержек, связанных с применением меню.

Элемент управления представляет собой окно особого вида, которое позволяет осуществлять специфический способ взаимодействия с пользователем. В качестве примеров таких окон можно привести кнопки, полосы прокрутки, окна редактирования и флажки (check boxes). Как и в случае со средствами меню, обработка элементов управления, которые определяются самой Windows, почти полностью автоматизирована. Поэтому ваша программа может использовать их без необходимости погружаться в рутину проработки всех деталей.

Диалоговое окно — специальное окно, которое позволяет осуществлять более сложное взаимодействие с программой по сравнению с возможностями, реализуемыми с помощью меню. Например, ваше приложение может использовать диалоговое окно, которое будет позволять пользователям вводить имя файла. Как правило, диалоговые окна содержат еще и элементы управления. В большинстве случаев ввод информации с помощью стандартных средств, реализуемый не через меню, осуществляется посредством диалогового окна.

Интерфейс прикладного программирования Win32

С точки зрения прикладного программиста, Windows 2000 (как и любая другая операционная система) характеризуется в первую очередь тем, каким именно образом программы взаимодействуют с ней. Все приложения “общаются” с Windows 2000 через *интерфейс, базирующийся на вызовах*. Базирующийся на вызовах интерфейс Windows 2000 — это весьма обширный набор системных функций, которые предоставляют доступ к функциональным возможностям операционной системы. В совокупности эти функции обозначаются термином Application Programming Interface (интерфейс прикладного программирования) или сокращенно — API. API содержит несколько сотен функций, которые могут использовать ваши прикладные программы, чтобы выполнять все необходимые операции для успешного взаимодействия с операционной системой. Например, распределение памяти, вывод информации на экран, создание окна и тому подобное. Подмножество функций API под названием GDI (Graphics Device Interface — графический интерфейс устройств) является той частью Windows, которая обеспечивает поддержку графического представления независимо от типа устройства.

Существует две основных разновидности API, получивших широкое распространение: Win16 и Win32. Win16 — более старая 16-разрядная версия API, которая исполь-

зуется операционной системой Windows 3.1. Win32 — современная 32-разрядная версия, интерфейс которой применяется программами в Windows 2000. (Win32 используется также системами Windows 95 и Windows 98.) В целом, Win32 охватывает множество функций Win16. На самом деле в большинстве случаев функции имеют одинаковые названия и применяются аналогичным образом. Тем не менее, будучи одинаковыми по сути и назначению, эти API отличаются друг от друга в двух фундаментальных аспектах. Во-первых, Win32 поддерживает 32-разрядную прямую адресацию, тогда как Win16 поддерживает только 16-разрядную сегментированную модель памяти. Это различие приводит к тому, что Win32, как правило, использует 32-разрядные значения аргументов и возвращаемых результатов в тех случаях, в которых Win16 применяет 16-разрядные значения. Во-вторых, Win32 включает функции API, которые поддерживают основанную на потоках многозадачность, защиту и другие продвинутые функциональные возможности, недоступные в Win16. В целом же не стоит слишком беспокоиться по поводу различий. Если вы новичок в Windows-программировании, эти различия если и затронут вас, то весьма незначительно. И то только в том случае, если вы будете переносить 16-разрядный программный код на платформу Windows 2000. Тогда вам необходимо будет просто внимательно проверить все аргументы, которые будут передаваться каждой функции API.

Компоненты окна

Перед тем как перейти к конкретным аспектам программирования в Windows 2000, необходимо объяснить несколько важных терминов. На рис. 26.1 представлено стандартное окно и его основные элементы, которые снабжены соответствующими надписями.

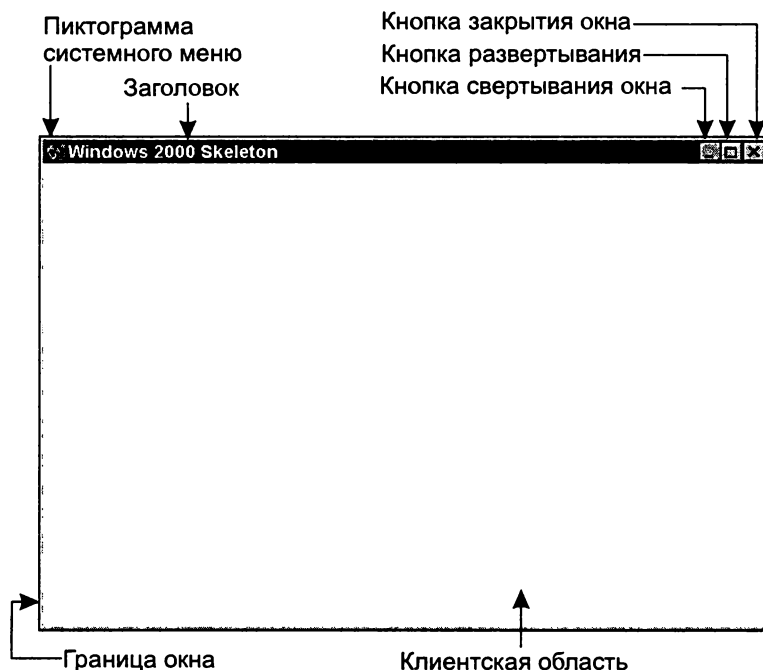


Рис. 26.1. Элементы стандартного окна.

Все окна имеют обрамление (рамку), которое определяет границы окна и используется для изменения его размеров. В верхней части окна расположено несколько элементов. В частности, в левом углу находится в виде кнопки значок системного меню (альтернативное название — значок панели заголовков). Щелчок на этой кнопке приводит к отображению на экране системного меню. Правее значка системного меню располагается заголовок окна. В правом углу находятся кнопки свертывания, разворачивания и закрытия окна. Клиентская область (часто употребляется такой термин как рабочая область) — это та часть окна, в которой происходит и отображается деятельность вашей программы. Кроме того, многие окна имеют горизонтальные и вертикальные полосы прокрутки, которые используются для прокрутки текста в пределах окна.

Взаимодействие прикладных программ с Windows

При написании программ для многих операционных систем обычно вы исходили из того, что именно ваша программа инициирует взаимодействие с данной операционной системой. Например, в системе DOS программа сама осуществляет запросы на выполнение таких операций, как ввод и вывод информации. Другими словами, программы, написанные “традиционным” способом, сами обращаются к операционной системе, а операционная система после запуска прикладную программу не вызывает. Windows же в значительной степени работает совсем наоборот. Именно Windows осуществляет обращение к прикладным программам. Этот процесс происходит примерно следующим образом. Программа находится в состоянии ожидания до тех пор, пока Windows не пошлет ей *сообщение*. Сообщение передается прикладной программе посредством специальной функции, которая вызывается самой Windows. После того как сообщение будет принято, предполагается, что прикладная программа должна выполнить соответствующее действие. Хотя в ответ на принятое сообщение прикладная программа может вызвать одну или несколько функций API, тем не менее, именно Windows инициирует всю эту “бурную” деятельность. По сравнению с остальными аспектами, именно механизм взаимодействия с Windows посредством сообщений больше всего определяет общий вид (структуру) всех Windows-программ.

Существует множество разнообразных типов сообщений, которые Windows 2000 может посылать вашей программе. Например, всякий раз, когда выполняется щелчок кнопкой мыши в пределах принадлежащего прикладной программе окна, ей (программе) посылается сообщение о щелчке кнопкой мыши. Другой тип сообщений посылается каждый раз, когда должно быть перерисовано окно, принадлежащее прикладной программе. Кроме того, сообщения иного вида посылаются прикладной программе каждый раз, когда пользователь нажимает клавишу, если приложение сфокусировано на вводе информации. Поэтому необходимо твердо усвоить следующую аксиому: построение прикладной программы должно исходить из предпосылки, что сообщения поступают к ней практически совершенно случайным образом. Вот почему Windows-приложения представляют собой управляемые прерываниями программы. Ведь вы не можете точно предсказать, каким будет следующее сообщение.

Базовые концепции функционирования приложений для Windows 2000

Прежде чем разрабатывать скелет приложения для Windows 2000, необходимо обсудить несколько базовых концепций, лежащих в основе всех Windows-программ.

WinMain()

Все Windows 2000-приложения начинают свою работу с вызова функции `WinMain()`. (В распоряжении Windows-программ отсутствует ее “не оконный” работающий по командам вариант — функция `main()`). `WinMain()` обладает специальными свойствами, которые отличают ее от других функций в вашем приложении. Во-первых, она должна быть скомпилирована в соответствии с WinAPI-соглашением о вызовах. По умолчанию функции используют C-соглашение о вызовах, но имеется возможность компилировать функцию так, чтобы она использовала другое соглашение. Например, широко распространенным вариантом является применение Pascal-соглашения о вызовах. По различным техническим причинам операционной системой Windows 2000 для вызова функции `WinMain()` применяется WinAPI-соглашение о вызовах. Результат, возвращаемый функцией `WinMain()`, должен иметь тип `int`.

Процедура окна

Все Windows-программы должны содержать специальную функцию, которая вызывается *не* вашей программой, а самой Windows. Эта функция обычно называется процедурой окна (*window procedure*) или функцией окна (*window function*). Именно посредством этой функции Windows 2000 взаимодействует с прикладными программами. Функция окна вызывается системой Windows 2000 в тех случаях, когда ей необходимо передать сообщение прикладной программе. Это сообщение функция окна принимает через свои параметры. Все функции окна должны быть объявлены таким образом, чтобы возвращаемое ими значение соответствовало типу `LRESULT CALLBACK`. Тип `LRESULT` представляет собой 32-разрядное целое число. Модель вызова функций `CALLBACK` используется в тех случаях, когда функцию вызывает сама Windows. В Windows-терминологии любая функция, вызываемая самой системой Windows, относится к классу функций *обратного вызова*.

К тому же, кроме получения сообщений, посланных операционной системой Windows 2000, функция окна должна инициировать выполнение действий, соответствующих содержащимся в сообщении указаниям. Как правило, в теле функции окна содержится оператор `switch`, который связывает конкретное ответное действие с тем сообщением, на которое программа будет реагировать. Прикладная программа не обязательно реагировать на все полученные сообщения. Те сообщения, которые не несут полезной информации для прикладной программы, вы можете отослать обратно в Windows 2000 для стандартной обработки, осуществляемой по умолчанию. Так как Windows может генерировать сотни различных сообщений, вполне естественно, что большинство сообщений обрабатывает Windows, а не прикладная программа.

Все сообщения представляют собой 32-разрядные целые значения. Кроме того, все сообщения сопровождаются некоторой дополнительной информацией, характерной для каждого сообщения.

Классы окон

Когда программа стартует под Windows 2000, первое, что ей необходимо сделать, так это определить и зарегистрировать *класс окна* (*window class*), который подразумевает *стиль* или *тип* этого окна. Когда прикладная программа регистрирует класс окна, она сообщает Windows сведения о форме (т. е. внешнем виде) и функции окна. Однако регистрация класса окна еще не приводит к его появлению на экране. Чтобы действительно создать окно, необходимо выполнить дополнительную работу.

Цикл обработки сообщений

Как уже упоминалось, Windows 2000 взаимодействует с прикладной программой посредством отправки ей сообщений. Все Windows-программы должны содержать цикл обработки сообщений внутри функции `WinMain()`. Этот цикл извлекает приходящее сообщение из очереди сообщений приложения, после чего отправляет его обратно в Windows. Операционная система, в свою очередь, вызывает функцию окна вашей программы, причем с этим же сообщением в качестве параметра. Это может показаться чересчур запутанным методом передачи сообщений, но, тем не менее, это единственный путь, которого должны придерживаться все Windows-программы. (Частично причина такого положения состоит в том, чтобы оперативно возвращать управление Windows. В этом случае планировщик заданий операционной системы может распределять время работы центрального процессора так, как он считает целесообразным, а не ожидать, когда закончится промежуток времени, выделенный для прикладной программы.)

Типы данных Windows

В функциях Windows API не очень часто используются стандартные типы данных языка программирования C, например `int` или `char*`. Вместо этого многие типы данных, применяемых в Windows, должны быть определены с помощью оператора `typedef` в файле `WINDOWS.H` и (или) в связанных с ним файлах. Этот файл поставляется компанией Microsoft (а также другими компаниями, выпускающими компиляторы C для платформы Windows) и должен включаться во все Windows-программы. Вот примеры наиболее употребительных типов данных: `HANDLE`, `HWND`, `UINT`, `BYTE`, `WORD`, `DWORD`, `LONG`, `BOOL`, `LPSTR` и `LPCSTR`. Тип `HANDLE` — это 32-разрядное целое число, которое используется как дескриптор. Существует множество типов дескрипторов, но все они имеют такой же размер, как и `HANDLE`. *Дескриптор* является просто значением, которое используется для уникальной идентификации некоторого объекта или ресурса. Например, тип `HWND` является 32-разрядным целым числом, которое используется в качестве дескриптора окна. К тому же, имена дескрипторов всех типов начинаются с буквы H. Тип `BYTE` является 8-разрядным целым без знака (`unsigned`); `WORD` — 16-разрядное короткое целое без знака; `DWORD` — 32-разрядное целое без знака; `UINT` — 32-разрядное целое без знака; `LONG` — 32-разрядное целое со знаком; `BOOL` — целый тип, используемый для указания величин, которые могут принимать значения, которые интерпретируются как `ИСТИНА` или `ЛОЖЬ`; `LPSTR` — указатель на строку, а тип `LPCSTR` — константный (`const`) указатель на строку.

В дополнение описанным выше базовым типам, в Windows 2000 определены еще несколько структур. Две из них, `MSG` и `WNDCLASSEX`, необходимы для создания скелета программы. Структура `MSG` содержит в себе сообщение Windows 2000, а `WNDCLASSEX` — структура, которая определяет класс окна. Позже в данной главе эти структуры будут обсуждаться более подробно.



Скелет программы для Windows 2000

Теперь, когда представлена вся необходимая предварительная информация, можно приступить к разработке минимального приложения для Windows 2000. Как уже говорилось, все программы для Windows 2000 имеют некоторые общие атрибуты. Скелет программы для Windows 2000, разработанный в этой главе, имеет все необходимые функциональные свойства. В мире Windows-программирования скелеты приложений (другими словами — программы-заготовки) используются довольно часто, поскольку “входная плата” при создании Windows-программ довольно значительна. В качестве

примера, сравните следующие показатели. В отличие от DOS-программ, у которых минимальный размер программы уложится всего в пять строк кода, минимальная программа для Windows составляет примерно пятьдесят строк.

Минимальная программа для Windows 2000 содержит две функции: WinMain() и функцию окна. Функция WinMain() должна выполнить следующие общие действия:

1. Описать класс окна;
2. Зарегистрировать этот класс в Windows 2000;
3. Создать окно данного класса;
4. Отобразить это окно;
5. Запустить выполнение цикла обработки сообщений.

Функция окна должна адекватно реагировать на все имеющиеся отношение к прикладной программе сообщения. Поскольку скелетная программа кроме отображения окна на экране дисплея больше ничего не делает, единственным сообщением, на которое она должна отреагировать, является сообщение о том, что пользователь прекратил выполнение программы.

Перед тем как перейти к подробному обсуждению отдельных вопросов, рассмотрим следующую программу, которая представляет собой минимальный скелет программы для Windows 2000. Эта программа-заготовка создает стандартное окно, содержащее заголовок, кнопки системного меню, а также стандартные кнопки свертывания, разворачивания и закрытия окна. Благодаря этому окно можно будет свернуть, развернуть, перемещать по экрану, изменять его размеры и, наконец, закрыть.

```
/* Минимальный скелет программы для Windows 2000. */

#include <windows.h>

LRESULT CALLBACK WindowFunc(HWND, UINT, WPARAM, LPARAM);

char szWinName[] = "MyWin"; /* имя класса окна */

int WINAPI WinMain(HINSTANCE hThisInst, HINSTANCE hPrevInst,
                  LPSTR lpszArgs, int nWinMode)
{
    HWND hwnd;
    MSG msg;
    WNDCLASSEX wcl;

    /* Определим класс окна. */
    wcl.cbSize = sizeof(WNDCLASSEX);

    wcl.hInstance = hThisInst; /* дескриптор данного
                               экземпляра */
    wcl.lpszClassName = szWinName; /* имя класса окна */
    wcl.lpfnWndProc = WindowFunc; /* функция окна */
    wcl.style = 0; /* стиль по умолчанию */
    /*
    wcl.hIcon = LoadIcon(NULL, IDI_APPLICATION); /* большая пиктограмма */
    wcl.hIconSm = NULL; /* использовать уменьшенный вариант
                        большой пиктограммы */
    wcl.hCursor = LoadCursor(NULL, IDC_ARROW); /* стиль
                                                курсора */

    wcl.lpszMenuName = NULL; /* класс меню отсутствует */
    wcl.cbClsExtra = 0; /* дополнительная память не
```

```

требуется */
wcl.cbWndExtra = 0;

/* Сделаем белым цвет фона окна. */
wcl.hbrBackground = (HBRUSH) GetStockObject(WHITE_BRUSH);

/* Зарегистрируем класс окна. */
if(!RegisterClassEx(&wcl)) return 0;

/* Поскольку класс окна уже зарегистрирован, теперь может
   быть создано окно. */
hwnd = CreateWindow(
    szWinName, /* имя класса окна */
    "Windows 2000 Skeleton", /* заголовок */
    WS_OVERLAPPEDWINDOW, /* стиль окна - стандартный */
    CW_USEDEFAULT, /* Координата X - пусть решает Windows */
    CW_USEDEFAULT, /* Координата Y - пусть решает Windows */
    CW_USEDEFAULT, /* Ширина - пусть решает Windows */
    CW_USEDEFAULT, /* Высота - пусть решает Windows */
    NULL, /* Дескриптор родительского окна - родительское
           окно отсутствует */
    NULL, /* Дескриптор меню - меню отсутствует */
    hThisInst, /* Дескриптор экземпляра */
    NULL /* Дополнительные аргументы отсутствуют */
);

/* Отобразим окно. */
ShowWindow(hwnd, nWinMode);
UpdateWindow(hwnd);

/* Создадим цикл обработки сообщений. */
while(GetMessage(&msg, NULL, 0, 0))
{
    TranslateMessage(&msg); /* трансляция клавиатурных сообщений */
    DispatchMessage(&msg); /* вернуть управление Windows 2000 */
}
return msg.wParam;
}

/* Эта функция вызывается Windows 2000 и пересылает
   сообщения из очереди сообщений.
*/
LRESULT CALLBACK WindowFunc(HWND hwnd, UINT message,
                             WPARAM wParam, LPARAM lParam)
{
    switch(message) {
        case WM_DESTROY: /* завершить программу */
            PostQuitMessage(0);
            break;
        default:
            /* Пусть Windows 2000 обрабатывает все сообщения, не
               перечисленные в предыдущем операторе switch. */
            return DefWindowProc(hwnd, message, wParam, lParam);
    }
    return 0;
}

```

Давайте тщательно, пункт за пунктом, проанализируем эту программу. Во-первых, все Windows-программы должны содержать заголовочный файл `WINDOWS.H`. Как уже упоминалось, этот файл (вместе с сопутствующими файлами) содержит прототипы функций API и всевозможные типы, макросы и описания, используемые самой Windows. Например, в файле `WINDOWS.H` (или в его придаточных файлах) определены типы данных `HWND` и `WNDCLASSEX`.

Функция окна, используемая данной программой, называется `WindowFunc()`. Она объявлена как функция обратного вызова, поскольку именно эту функцию Windows вызывает для взаимодействия с данной программой.

Как уже говорилось, работа программы начинается с выполнения `WinMain()`. Функции `WinMain()` передается четыре параметра. Из них `hThisInst` и `hPrevInst` — дескрипторы. Дескриптор `hThisInst` относится к текущему экземпляру программы. Помните, что Windows 2000 является многозадачной системой, поэтому одновременно может выполняться более одного экземпляра вашей программы. Для Windows 2000 дескриптор `hPrevInst` всегда принимает значение `NULL`. Параметр `lpzArgs` является указателем на строку, которая содержит аргументы командной строки, указанные при запуске приложения. В Windows 2000 эта строка содержит всю командную строку, в том числе и имя программы. Параметр `nWinMode` содержит значение, которое определяет то, как будет отображаться окно в момент, когда программа начнет выполняться.

При выполнении данной функции в ней будут созданы три переменные. Переменная `hwnd` будет содержать дескриптор окна программы. Структурная переменная `msg` будет содержать сообщение окна, а структурная переменная `wcl` будет использоваться для описания класса окна.

Определение класса окна

В первую очередь функция `WinMain()` выполняет два действия: определение класса окна, а затем его регистрация. Класс окна описывается путем заполнения необходимых значений в полях, определяемых структурой `WNDCLASSEX`. Вот эти поля:

```
UINT cbSize;           /* размер структуры WNDCLASSEX */
UINT style;            /* тип окна */
WNDPROC lpfnWndProc;   /* адрес функции окна */
int cbClsExtra;        /* дополнительная память класса */
int cbWndExtra;        /* дополнительная память окна */
HINSTANCE hInstance;   /* дескриптор данного экземпляра */
HICON hIcon;           /* дескриптор большой пиктограммы */
HICON hIconSm;        /* дескриптор маленькой пиктограммы */
HCURSOR hCursor;      /* дескриптор указателя мыши */
HBRUSH hbrBackground; /* цвет фона */
LPCSTR lpszMenuName;   /* имя главного меню */
LPCSTR lpszClassName; /* имя класса окна */
```

Как видно из приведенного листинга, `cbSize` задает размер структуры `WNDCLASSEX`. Элемент `hInstance` определяет дескриптор текущего экземпляра и устанавливается в соответствии со значением дескриптора `hThisInst`. Имя класса окна указывается с помощью поля `lpszClassName`, которое в нашем случае указывает на строку `"MyWin"`. Адрес функции окна устанавливается в `lpfnWndProc`. В данной программе не назначается стиль по умолчанию, не требуется никакой дополнительной информации и не определяется главное меню. Хотя большинство программ содержат главное меню, в нем нет никакой необходимости для скелета приложения.

Все Windows-приложения должны определять используемые по умолчанию формы (изображения) указателя мыши и пиктограммы приложения. Прикладная программа может определять свою собственную пользовательскую версию этих ресурсов или она может использовать один из встроенных стилей, что и создает скелет приложения. В любом случае дескрипторы этих ресурсов должны быть присвоены соответствующим элементам структуры WNDCLASSEX. Чтобы лучше понять, как это делается, начнем, пожалуй, с пиктограмм.

Любое приложение для Windows 2000 имеет две связанные с ним пиктограммы: одна большая и одна маленькая. Маленькая пиктограмма используется в том случае, когда окно приложения свернуто. Эта же пиктограмма используется для отображения значка системного меню программы. Большая пиктограмма отображается на экране в том случае, когда вы перемещаете или копируете свое приложение на рабочий стол Windows. Как правило, большие пиктограммы представляют собой растровые изображения размером 32×32 пикселя, а маленькие — размером 16×16 пикселей. Большая пиктограмма загружается посредством API-функции `LoadIcon()`, чей прототип приведен ниже:

```
HICON LoadIcon(HINSTANCE hInst, LPCSTR lpszName);
```

Эта функция возвращает дескриптор пиктограммы, а в случае аварийного завершения — значение `NULL`. В приведенном примере *hInst* определяет дескриптор модуля, который содержит пиктограмму, а ее название определяется параметром *lpszName*. Впрочем, чтобы воспользоваться одной из встроенных пиктограмм, необходимо использовать `NULL` для первого параметра и указать один из следующих макросов в качестве второго параметра:

Макрос пиктограммы
`IDI_APPLICATION`
`IDI_ERROR`
`IDI_INFORMATION`
`IDI_QUESTION`
`IDI_WARNING`
`IDI_WINLOGO`

Картинка
Пиктограмма по умолчанию
Символ ошибки
Символ информации
Знак вопроса
Восклицательный знак
Логотип Windows

При загрузке пиктограмм следует обратить особое внимание на два важных момента. Во-первых, если ваше приложение не определяет маленькую пиктограмму, будет исследован ресурсный файл большой пиктограммы. Если в нем содержится маленькая пиктограмма, то именно она и будет использована. В противном случае при необходимости маленькая пиктограмма будет получена в результате уменьшения (пропорционального сжатия) большой пиктограммы. Если вы не хотите определять маленькую пиктограмму, присвойте значение `NULL` параметру *hIconSm* — именно так и поступает наша программа-заготовка. Во-вторых, функция `LoadIcon()`, вообще говоря, может применяться только для загрузки большой пиктограммы. Для загрузки пиктограмм произвольного размера можно воспользоваться функцией `LoadImage()`.

Чтобы загрузить указатель мыши, используйте API-функцию `LoadCursor()`. Данная функция имеет следующий прототип:

```
HCURSOR LoadCursor(HINSTANCE hInst, LPCSTR lpszName);
```

Эта функция возвращает дескриптор ресурса курсора или `NULL` в случае аварийного завершения. В данном примере *hInst* определяет дескриптор модуля, содержащего курсор мыши, а его имя указывается в параметре *lpszName*. Чтобы воспользоваться одним из встроенных указателей мыши, необходимо использовать `NULL` в качестве первого параметра и задать макрос одного из встроенных курсоров мыши в качестве второго параметра. Ниже приведено несколько встроенных курсоров:

Макрос курсора мыши

IDC_ARROW
IDC_CROSS
IDC_HAND
IDC_IBEAM
IDC_WAIT

Форма

Указатель-стрелка по умолчанию
Перекрестие
Рука
Вертикальная двутавровая балка
Песочные часы

В качестве цвета фона окна, созданного скелетом программы, выбран белый цвет, а дескриптор *кисти* (*brush*) получается с помощью API-функции `GetStockObject()`. Кисть является ресурсом, который окрашивает экран с учетом предварительно заданных размера, цвета и узора. Функция `GetStockObject()` применяется для получения дескриптора ряда стандартных объектов отображения, в том числе кистей, перьев (которые проводят линии) и шрифтов символов. Вот его прототип:

```
HGDIOBJ GetStockObject(int object);
```

Данная функция возвращает дескриптор объекта, определенного параметром *object*. В случае аварийного завершения возвращается значение `NULL`. (Тип `HGDIOBJ` относится к `GDI`-дескрипторам). Ниже приведено несколько встроенных кистей, доступных вашей программе:

Имя макроса

BKACK_BRUSH
DKGRAY_BRUSH
HOLLOW_BRUSH
LTGRAY_BRUSH
WHITE_BRUSH

Тип фона

Темно серый
Полупрозрачный (видно сквозь окно)
Черный
Светло серый
Белый

Для получения кисти можно использовать эти макросы в качестве параметров функции `GetStockObject()`.

После того как класс окна полностью определен, он регистрируется в Windows 2000 с помощью API-функции `RegisterClassEx()`, прототип которой приведен ниже:

```
ATOM RegisterClassEx(CONST WNDCLASSEX *lpWClass);
```

Эта функция возвращает значение, которое идентифицирует класс окна. `ATOM` является `typedef`-описанием типа, которое подразумевает тип `WORD`. Каждый класс окна принимает уникальное значение. Параметр *lpWClass* должен содержать адрес структуры `WNDCLASSEX`.

Создание окна

После того, как класс окна определен и зарегистрирован, ваше приложение может на самом деле создать окно этого класса с помощью API-функции `CreateWindow()`, прототип которой выглядит следующим образом:

```
HWND CreateWindow(  
    LPCSTR lpszClassName, /* название класса окна */  
    LPCSTR lpszWinName,   /* заголовок окна */  
    Dword dwStyle,        /* тип окна */  
    int X, int Y,         /* координаты верхней левой точки */  
    int Width, int Height, /* размеры окна */  
    HWDN hParent,        /* дескриптор родительского окна */  
    HMENU hMenu,         /* дескриптор главного меню */  
    HINSTANCE hThisInst,  /* дескриптор создателя */  
    LPVOID lpszAdditional, /* указатель на дополнительную  
                           информацию */  
);
```

Как видно из листинга скелета программы, многим параметрам функции `CreateWindow()` значения могут присваиваться по умолчанию или же в качестве значения им можно присвоить `NULL`. В действительности, в качестве параметров *X*, *Y*, *Width* и *Height* чаще всего используется макрос `CW_USEDEFAULT`; в этом случае Windows 2000 выбирает подходящий размер и местоположение окна. Если данное окно не имеет родительского окна (а именно этот случай имеет место в нашем скелете программы), то в качестве параметра *hParent* может быть указан `NULL`. (Для указания значения этого параметра можно также использовать `HWND_DESKTOP`.) Если окно не содержит главного меню или использует главное меню, которое определено посредством класса окна, то параметр *hMenu* должен иметь значение `NULL`. (Параметр *hMenu* имеет также и другие применения.) К тому же, если никакая дополнительная информация не требуется, что характерно для большинства случаев, то параметру *lpszAdditional* можно присвоить значение `NULL`. (Тип `LPCVOID` переопределяется оператором `typedef` как `void*`. Исторически сложилось так, что `LPCVOID` обозначает длинный указатель на `void`.)

Значения остальных четырех параметров должны быть явно установлены прикладной программой. Во-первых, параметр *lpszClassName* должен указывать на имя класса окна. (Это то имя, которое вы дали окну при его регистрации.) Заголовок окна — это последовательность символов, на которую указывают посредством *lpszWinName*. Это может быть и пустая строка, но, как правило, окну следует давать какой-то заголовок. Стил (или тип) окна, созданного в действительности, определяется значением параметра *dwStyle*. Макрос `WS_OVERLAPPEDWINDOW` определяет стандартное окно, которое имеет системное меню, обрамление и кнопки свертывания, разворачивания и закрытия окна. Хотя чаще всего используется именно такой стиль окна, вы можете построить окно, удовлетворяющее вашим собственным критериям. Для этого просто объедините с помощью оператора `OR` макросы различных необходимых вам стилей. Ниже приведены некоторые часто встречающиеся стили:

Макрос стиля

`WS_OVERLAPPED`
`WS_MAXIMIZEBOX`
`WS_MINIMIZEBOX`
`WS_SYSMENU`
`WS_HSCROLL`
`WS_VSCROLL`

Функция Windows

Перекрывающееся окно с обрамлением
 Кнопка разворачивания
 Кнопка свертывания
 Системное меню
 Горизонтальная полоса прокрутки
 Вертикальная полоса прокрутки

Параметр *hThisInst* игнорируется операционной системой Windows 2000, но для Windows 95/98 он должен содержать дескриптор текущего экземпляра приложения. Поэтому, чтобы обеспечить совместимость с этими средами, а заодно и предотвратить проблемы в будущем, параметру *hThisInst* рекомендуется присваивать значение дескриптора текущего экземпляра, как это сделано в нашем скелете программы.

Функция `CreateWindow()` возвращает дескриптор созданного ею окна или `NULL`, если окно не может быть создано.

Даже после создания окна оно все еще не отображается на экране дисплея. Чтобы отобразить окно, надо вызвать API-функцию `ShowWindow()`. Эта функция имеет следующий прототип:

```
BOOL ShowWindow(HWND hwnd, int nHow);
```

Дескриптор отображаемого дисплеем окна указывается в параметре *hwnd*. А параметром *nHow* определяется режим отображения. Если окно выводится на экран в первый раз, целесообразно в качестве параметра *nHow* указать значение параметра `nWinMode` функции `WinMain()`. Значение `nWinMode` определяет способ отображения окна сразу после запуска программы на выполнение. Последующие вызовы могут при необходимости отобразить окно в нужном виде или вовсе удалить его. Некоторые общепотребительные значения параметра *nHow* приведены ниже:

Макрос отображения

SW_HIDE
SW_MINIMIZE
SW_MAXIMIZE
SW_RESTORE

Получаемый эффект

Удаляет окно с экрана
Свертывает окно в пиктограмму
Развертывает окно
Возвращает окну обычный размер

Функция `ShowWindow()` возвращает статус предыдущего режима отображения окна. Если окно выводилось на экран, то возвращается ненулевое значение. А если окно не отображалось на экране, то возвращается нуль.

Хотя с формальной точки зрения для скелета программы это и не обязательно, все же в его текст включен вызов функции `UpdateWindow()`, поскольку она необходима практически для всех Windows 2000-приложений. По существу, она указывает, что Windows 2000 должна послать вашему приложению сообщение о том, что его главное окно необходимо обновить.

Цикл обработки сообщений

Финальная часть функции `WinMain()` заготовки прикладной программы относится к циклу *обработки сообщений*. Цикл обработки сообщений является составной частью всех Windows-приложений. Его назначение — принять и обработать сообщение, посланное Windows 2000. Во время выполнения прикладной программы ей постоянно посылаются сообщения. Все эти сообщения сохраняются в очереди сообщений приложения и находятся там до тех пор, пока они не будут извлечены и обработаны. Всякий раз, когда приложение готово извлечь следующее сообщение, оно должно вызвать API-функцию `GetMessage()`, имеющую следующий прототип:

```
BOOL GetMessage(LPMSG msg, HWND hwnd, UINT min, UINT max);
```

Сообщение будет записано в структуру, на которую указывает параметр `msg`. Все сообщения Windows имеют тип структуры `MSG`, представленный ниже:

```
/* Структура сообщения */
typedef struct tagMSG
{
    HWND hwnd;           /* окно, для которого предназначено сообщение */
    UINT message;        /* сообщение */
    WPARAM wParam;       /* информация, обусловленная сообщением */
    LPARAM lParam;       /* дополнительная информация, обусловленная
                          сообщением */
    DWORD time;          /* время, когда было отправлено сообщение */
    POINT pt;            /* координаты X и Y местоположения указателя мыши */
} MSG;
```

В структуре `MSG` дескриптор окна, которому предназначается сообщение, содержится в `hwnd`. Все сообщения в Windows 2000 являются 32-разрядными целыми числами, а само сообщение содержится в поле `message`. В зависимости от конкретного сообщения обусловленная им дополнительная информация передается в `wParam` и `lParam`. Оба типа `WPARAM` и `LPARAM` являются 32-разрядными значениями.

Время, когда было отправлено (зарегистрировано) сообщение, определяется в миллисекундах в поле `time`.

Элемент `pt` содержит координаты указателя мыши в тот момент, когда было отправлено сообщение. Координаты хранятся в структуре `POINT`, которая определяется следующим образом:

```
typedef struct tagPOINT {
    LONG x, y;
} POINT;
```

Если в очереди сообщений приложения отсутствуют сообщения, то вызов функции GetMessage() приведет к передаче управления обратно Windows 2000.

Параметр *hwnd* функции GetMessage() определяет, для какого окна будут получены сообщения. Ведь часто приложение имеет несколько окон, и иногда необходимо принимать сообщения только для конкретного окна. Ну а если этому параметру присвоить значение NULL, то в ваше приложение будут направляться все сообщения.

Оставшиеся два параметра функции GetMessage() определяют диапазон получаемых сообщений. Обычно приложение должно принимать все сообщения. Для этого необходимо оба параметра (и *min*, и *max*) установить равными нулю; именно так и сделано в скелете программы.

Функция GetMessage() возвращает нуль, когда пользователь прекращает работу программы, что приводит к завершению цикла обработки сообщений. Во всех остальных случаях данная функция возвращает ненулевое значение. Если происходит ошибка, эта функция возвращает -1. Ошибка может произойти только при необычных обстоятельствах, которые не приемлемы для работы большинства программ.

Внутри цикла обработки сообщений осуществляется вызов двух функций. Первая — это API-функция TranslateMessage(). Эта функция транслирует генерируемые операционной системой Windows 2000 виртуальные коды клавиш в символьные сообщения. Хотя ее применение необязательно во всех приложениях, в большинстве из них все же используется вызов функции TranslateMessage(), поскольку она необходима для осуществления полной интеграции клавиатуры в вашу прикладную программу.

После того как сообщение было извлечено и преобразовано, оно отсылается обратно в Windows 2000 с помощью API-функции DispatchMessage(). Затем Windows 2000 хранит это сообщение до тех пор, пока не сможет передать его функции окна вашей программы.

Как только завершается цикл обработки сообщений, выполнение функции WinMain() заканчивается, при этом она возвращает Windows 2000 значение msg.wParam. Это значение содержит код возврата, генерируемый при завершении выполнения прикладной программы.

Функция окна

Второй функцией в скелете приложения является его функция окна. В нашем случае эта функция названа WindowFunc(), хотя она может носить любое понравившееся вам имя. Сообщения передаются функции окна посредством Windows 2000. Первые четыре элемента структуры MSG являются его параметрами. Из них единственным параметром, который используется скелетом программы, является собственно сообщение.

Функция окна нашей программы-заготовки реагирует явным образом только на одно сообщение: WM_DESTROY. Такое сообщение посылается тогда, когда пользователь прекращает работу приложения. После получения такого сообщения программа должна осуществить вызов API-функции PostQuitMessage(). Аргументом этой функции является код возврата, который помещается в msg.wParam внутри функции WinMain(). Вызов PostQuitMessage() приводит к передаче приложению сообщения WM_QUIT, что заставляет функцию GetMessage() вернуть значение ЛОЖЬ (false) и, следовательно, прекратить работу вашей программы.

Все остальные сообщения, принимаемые функцией WindowFunc(), передаются операционной системе Windows 2000 посредством вызова DefWindowProc() для стандартной обработки по умолчанию. Этот этап необходим, поскольку все сообщения так или иначе должны быть обработаны.

Каждое сообщение устанавливает некоторое значение, которое должно быть возвращено функцией окна после обработки сообщения. Обычно после обработки большей части сообщений необходимо возвращать нулевое значение. Но после обработки некоторых сообщений требуется вернуть другой код возврата.

Файл описания больше не нужен

Если вам приходилось создавать 16-разрядные программы для Windows, то вы помните, что при этом необходимо было использовать еще и *файлы описаний* (*definition files*). Что касается 16-разрядных версий Windows, все программы должны были иметь сопутствующие файлы описаний. Файл описаний — это просто текстовый файл, который задает определенную информацию и параметры, которые необходимо указывать для 16-разрядной среды. Поскольку Windows 2000 имеет 32-разрядную архитектуру (а также другие усовершенствования), файлы описаний, как правило, не являются больше необходимым атрибутом для программ, работающих под управлением Windows 2000. Если вы новичок в программировании для Windows и не имеете никакого представления о файлах описаний, то следующий материал будет вам, конечно же, полезен.

Все файлы описаний имеют расширение .DEF. Например, файл описаний для нашей программы-заготовки мог бы иметь название SKELL.DEF. Ниже приведен файл описания, который можно использовать для обеспечения нисходящей совместимости с Windows 3.1:

```
DESCRIPTION 'Skeleton Program'
EXETYPE WINDOWS
CODE PRELOAD MOVEABLE DISCARDABLE
DATA PRELOAD MOVEABLE MULTIPLE
HEAPSIZE 8192
STACKSIZE 8192
EXPORTS WindowFunc
```

Приведенный файл определяет название программы и ее описание, причем оба эти параметра не являются обязательными. Здесь также формулируется утверждение, что исполняемый файл будет совместим с Windows (а не с DOS, например). Оператор CODE сообщает Windows 2000, что при запуске программы ее необходимо загрузить всю целиком (элемент PRELOAD), что код программы разрешается перемещать в памяти (элемент MOVEABLE) и что данный код может быть удален из памяти и загружен повторно, когда в этом возникнет необходимость (DISCARDABLE). К тому же, этот файл определяет то, что данные для прикладной программы должны быть загружены в самом начале ее выполнения, а также допускается их перемещение в другие области памяти. Здесь же устанавливается, что каждый экземпляр программы имеет свои собственные данные (MULTIPLE). Следующие строки сообщают, что программе выделяются динамически распределяемая область памяти (куча) и стек указанных размеров. И, наконец, указывается имя экспортируемой функции окна. Экспорт функции предоставляет возможность системе Windows 3.1 осуществлять ее вызов.

Не забывайте, что файлы описаний редко используются в процессе программирования для 32-разрядных версий Windows.

Соглашения об именовании

Прежде чем закончить данную главу, необходимо привести небольшие пояснения по поводу именования функций и переменных. Новичкам в Windows-программировании некоторые имена переменных и параметров в программе-заготовке и ее описании покажутся, вероятно, довольно необычными. Причина кроется в том, что подобные имена являются следствием строгого соблюдения ряда соглашений о присвоении имен, которые были изобретены и учреждены компанией Microsoft для программирования под управлением Windows. Согласно этим соглашени-

ям, название функции состоит из глагола, за которым следует существительное. Причем первая буква и глагола, и существительного пишутся с большой буквы.

Что касается имен переменных, то здесь Microsoft избрала путь применения довольно сложной системы указания типа данных в имени переменной. В соответствии с ней, впереди имени переменной добавляется префикс типа (который пишется строчными буквами). Непосредственно имя переменной пишется с заглавной буквы. Префиксы типов приведены в табл. 26.1. Целесообразность применения префиксов типа не является очевидной (для многих она кажется скорее даже спорной), поэтому такая модель именования не получила повсеместного признания и не стала универсальной. Многие Windows-программисты пользуются именно данным способом именования, но в то же время не меньшее количество не приемлют его. Вам, естественно, предоставляется свобода выбора применять любое соглашение об именованиях, которое придется вам по душе.

Таблица 26.1. Символы префикса для переменных различных типов

<i>Префикс</i>	<i>Тип данных</i>
b	булев (1 байт)
c	символ (1 байт)
dw	длинное целое без знака
f	16-разрядное битовое поле (флаги)
fn	функция
h	дескриптор
l	длинное целое
lp	длинный указатель
n	короткое целое
p	указатель
pt	длинное целое, содержащее координаты экрана
w	короткое целое без знака
sz	указатель на строку — массив, оканчивающийся нулевым символом
lpsz	длинный указатель на строку — массив, оканчивающийся нулевым символом
rgb	длинное целое, содержащее значения RGB-цветов

Полный
справочник по



Глава 27

**Проектирование программ
с помощью C**

Начнем главу со сравнений. Так, создание большой компьютерной программы и проектирование крупного здания имеют много общих черт. В английском языке даже должность разработчика программного обеспечения в наши дни часто называется “architect” (“архитектор”). Без сомнения, скрытые от взора непосвященных механизмы, благодаря которым делается возможным как проектирование крупных зданий, так и проектирование крупных программ, в действительности подобны друг другу. И заключаются они в применении подходящих методов проектирования. В этой главе будут подробно рассмотрены некоторые методы проектирования, характерные для среды программирования на языке С; эти методы значительно облегчают процесс разработки и сопровождения программ.

Данная глава в основном предназначена для новичков в программировании. Опытным профессионалам большая часть изложенного здесь материала будет, конечно, уже знакома.

Проектирование сверху вниз

Без сомнения, главнейшее условие успешного создания крупных программ заключается в применении надежных методов проектирования. Широкое распространение при написании программ получили следующие три метода: нисходящий (сверху вниз), восходящий (снизу вверх) и специальный (на данный конкретный случай). В случае *нисходящего метода* вы начинаете созидательный процесс с программы высокого уровня и спускаетесь до подпрограмм низкого уровня. *Восходящий метод* работает в обратном направлении: вы начинаете с отдельных специальных подпрограмм, постепенно строите на их основе более сложные конструкции и заканчиваете самым верхним уровнем программы. *Специальный подход* не имеет заранее установленного метода.

Поскольку С является структурированным языком программирования, то лучше всего он сочетается с нисходящим методом проектирования. Подход сверху вниз позволяет производить ясный, легко читаемый программный код, который в дальнейшем не вызовет трудностей и при сопровождении. К тому же данный подход помогает прояснить и сделать понятной всю структуру программы в целом до кодирования функций более низкого уровня. Такой подход позволяет уменьшить потери времени, обусловленные неудачными или ошибочными начинаниями.

Структурирование программы

Как и для любой общей схемы, при применении нисходящего метода начинают с общего описания программы, а затем переходят к проработке ее конкретных элементов. На практике при разработке программы лучше всего сначала точно определить, что программа будет делать на самом высоком уровне, и только после этого погружаться в подробности, касающиеся каждого действия. Предположим, к примеру, что необходимо написать программу, предназначенную для ведения списка рассылки. Во-первых, необходимо составить перечень действий, которые будет выполнять такая программа. Каждая строчка данного перечня должна содержать только один функциональный элемент. (На этом этапе под функциональным элементом понимается “черный ящик”, который выполняет только одну задачу.) Тогда такой перечень может быть представлен примерно в следующем виде:

- Ввести новый адрес
- Удалить новый адрес
- Печать списка
- Найти имя

- Сохранить список
- Загрузить список
- Завершить выполнение программы

После того как вы определили все функциональные возможности программы, можно в общих чертах описать подробные свойства каждого функционального модуля, начиная с основного цикла. Ниже приведен один из возможных вариантов такого представления, позволяющий реализовать основной цикл программы работы со списком рассылки:

```
main loop
{
    do {
        вывести меню
        определить выбор пользователя
        выполнить выбранное действие
    } while выбор != quit
}
```

Такая разновидность алгоритмического представления (иногда называемая *псевдокодом*) может помочь внести ясность в общую структуру программы; ее необходимо выполнить до кодирования. С-подобный синтаксис был использован по той простой причине, что он вам уже знаком, но можно воспользоваться и любым другим подходящим синтаксисом.

Аналогичные определения необходимо дать каждому функциональному элементу. Например, вы можете описать функцию, которая осуществляет запись списка рассылки в файл на диске примерно следующим образом:

```
save to disk {
    open disk file
    while есть данные write {
        write данные на диск
    }
    close disk file
}
```

На этом уровне абстракции функция “записать-на-диск” обращается к новым функциональным модулям более низкого уровня. Эти модули открывают файл на диске, записывают на него данные, а затем закрывают дисковый файл. В дальнейшем необходимо будет еще определить и каждый из этих модулей. Если при их описании будут создаваться новые функциональные элементы, они также должны быть определены и т.д. Этот процесс закончится, когда при описании больше не будет создан ни один новый функциональный элемент. Тогда остается всего лишь сесть и написать реальный С-код, который реализует эти действия. Например, модуль, который закрывает дисковый файл, вероятно, будет транслирован в вызов функции `fclose()`.

Обратите внимание, что при подобном определении совсем ничего не упоминается о структуре данных и переменных. Это сделано умышленно, потому что до сих пор вы хотели определить только то, что должна делать ваша программа, но не то, каким образом она реально будет это осуществлять. Такой описательный процесс помогает выбрать эффективную структуру данных. (Естественно, структуру данных необходимо будет описать перед кодированием функциональных элементов.)

Выбор структуры данных

После определения общей структуры прикладной программы необходимо решить, какой будет структура данных. Выбор структуры данных и ее реализация имеют чрезвычайно важное значение, поскольку она помогает определить проектные ограничения вашей программы.

Список рассылки, как правило, содержит следующую информацию: имена, названия улиц, городов, штатов и почтовые индексы. При нисходящем подходе это немедленно предполагает применение определенной структуры для хранения этой информации. И тут же возникает вопрос: а как такие структуры будут храниться и обрабатываться? Для программы, работающей со списком рассылки, можно было бы использовать массив структур фиксированного размера. Но массив фиксированного размера имеет один серьезный недостаток: размер массива жестко и безапелляционно ограничивает длину списка рассылки. Более удачное решение заключается в динамическом выделении памяти для каждого адреса. Тогда каждый адрес будет сохраняться в динамической структуре данных определенной формы (например, в связанном списке), которая может при необходимости расти или уменьшаться. В таком случае список может быть огромным или очень маленьким в зависимости от конкретных обстоятельств.

Хотя на данном этапе мы отдали предпочтение динамическому распределению памяти, а не массиву фиксированного размера, все же точная модель представления данных все еще не определена. Существует несколько возможных вариантов. Можно использовать однонаправленный связный список, двунаправленный связный список, двоичное дерево и даже метод хэширования. Каждый из этих методов имеет свои достоинства и недостатки. Ограничим круг обсуждения и предположим, что конкретно нашему приложению, обрабатывающему список рассылки, предъявляется требование достичь минимального времени поиска. Исходя из этого, мы выбираем двоичное дерево. Теперь можно определить структуру, которая содержит в списке каждое имя и адрес, как показано ниже:

```
struct addr {  
    char name[30];  
    char street[40];  
    char city[20];  
    char state[3];  
    char zip[11];  
    struct addr *left; /* указатель на левое поддерево */  
    struct addr *right; /* указатель на правое поддерево */  
};
```

Теперь, после определения структуры данных, можно приступить к кодированию всей программы. Для этого надо всего лишь определить все подробности, описанные в общей структуре созданного ранее псевдокода. Если вы последуете нисходящему методу, программы будут не только легко читаться, но и потребуют меньше времени на разработку и сопровождение.



“Пуленепробиваемые” функции

В больших программах, особенно в тех, которые предназначены для управления устройствами, которые потенциально могут представлять угрозу для жизни, любая возможность возникновения ошибки должна быть сведена к минимуму. Маленькие программы еще могут быть верифицированы (тщательно проверены на отсутствие ошибок), но с большими программами такая работа вряд ли осуществима. (Верифицированная программа не содержит ошибок и никогда не допускает сбоев в работе, по крайней мере, теоретически.) К примеру, представьте себе программу, которая управляет закрылками современного реактивного самолета. Физически невозможно протестировать все возможные взаимодействия сил, которые могут быть приложены к самолету. Это означает, что вы не можете в полной мере протестировать программу. В лучшем случае, максимум, что можно будет сказать о ней — она работает корректно в таких-то условиях и при таких-то обстоятельствах. В программах по-

добного рода вы (как пассажир или программист) вряд ли захотели бы столкнуться с крушением (программы или самолета)!

После программирования в течение нескольких лет вы заметите, что хотя большинство отказов в работе программы происходят по причине самых разных ошибок программирования, само количество типов этих ошибок сравнительно невелико. Например, многие так называемые катастрофические программные сбои вызваны одной из следующих довольно частых ошибок:

- Какое-то условие привело к непредусмотренному выполнению бесконечного цикла;
- Были нарушены границы массива, что привело к повреждению примыкающего к нему кода программы или данных;
- Неожиданное переполнение при обработке данных определенного типа.

Теоретически при наличии необходимого опыта программирования ошибок подобного рода можно избежать в процессе тщательного продумывания и внимательной реализации проекта. (И в самом деле, профессионально написанные программы не должны содержать ошибок подобного рода.)

Тем не менее, после первого этапа разработки программ часто появляются ошибки иного типа, проявляющиеся или во время заключительного процесса “тонкой настройки”, или на стадии сопровождения программы. Такие ошибки возникают из-за того, что одна функция по непредусмотренным причинам влияет на код или данные другой функции. Подобные ошибки обнаружить исключительно тяжело, поскольку код обеих функций может оказаться вполне корректным, а появление ошибки вызывает именно взаимодействие этих функций. Поэтому вполне естественно, что стремление уменьшить вероятность появления катастрофических сбоев приводит к желанию сделать свои функции и их данные как можно более “пуленепробиваемыми”. Наилучший способ достижения этой цели состоит в том, чтобы код и данные каждой функции были скрыты как друг от друга, так и от остальной части программы.

Методы сокрытия кода и данных во многом аналогичны механизму передачи секретной информации только тем лицам, кому ее необходимо знать. Проще говоря, если какая-то функция не должна знать что-либо о другой функции или переменной, не предоставляйте этой функции возможности доступа к ним. Для этого необходимо придерживаться следующих четырех правил-принципов:

1. Каждый функциональный элемент должен иметь только одну точку входа и одну точку выхода;
2. Везде, где только возможно, не используйте глобальных переменных, а явно передавайте информацию функциям (например, посредством параметров);
3. В случаях, когда в нескольких зависимых функциях используются глобальные переменные, необходимо размещать как эти переменные, так и функции в обособленном файле. К тому же, в таких случаях глобальные переменные должны быть объявлены как `static`;
4. Каждая функция должна сообщать вызвавшей ее программе об успешном или аварийном завершении намеченной ей операции. То есть вызывающая программа должна распознавать признаки успешного или сбойного завершения функции.

Правило 1 устанавливает, что каждая выполняемая функция имеет только одну точку входа и одну точку выхода. Это означает, что хотя в функциональный элемент может входить несколько функций, остальная часть программы взаимодействует только с одной из них. Обратимся к программе, обрабатывающей список рассылки; эта программа обсуждалась в предыдущих разделах. В ней предусмотрено выполнение семи функций. Можно поместить каждую функцию, вызываемую из соответствующего функционального поля, в свой собственный файл и компилировать их все отдельно друг от друга. Если все будет сделано надлежащим образом, то единственной точкой

входа и выхода каждого функционального элемента будет его функция наиболее высокого уровня. А применительно к программе, обрабатывающей список рассылки, это означает, что такие высокоуровневые функции будут вызываться только функцией `main()`, тем самым будет предотвращено случайное разрушение одного функционального элемента другим. Эту ситуацию поясняет рис.27.1.

Наилучший способ уменьшения вероятности появления побочных эффектов состоит в том, чтобы всегда явно передавать конкретной функции всю необходимую ей информацию. Правда, такое решение в некоторых случаях может ухудшить параметры производительности. Тем не менее, во всех случаях старайтесь избегать применения глобальных данных. Это уже *правило 2*, и если вы когда-нибудь писали крупные программы с помощью стандартной версии BASIC (в которой все переменные глобальные), то вы, наверное, уже понимаете важность соблюдения этого принципа.

Правило 3 устанавливает, что в тех случаях, когда все же необходимо применить глобальные данные, то сами они и те функции, которые обращаются к ним, должны быть размещены в одном файле и компилироваться отдельно от остальной части приложения. Главный принцип здесь состоит в том, чтобы объявлять глобальные данные как `static`, тогда они будут доступны из других файлов. К тому же, функции, осуществляющие доступ к данным типа `static`, могут быть сами объявлены как `static`, что предохранит их от вызова функциями, которые не были объявлены в том же самом файле.

Правило 4, попросту говоря, гарантирует, что программы получают “вторую попытку”, так как программа, вызвавшая определенную функцию, может приемлемым образом реагировать на ситуацию, в которой возникла ошибка. Например, допустим, что при выполнении функции, осуществляющей управление закрылками самолета, непредвиденно происходит выход за диапазон представимых значений. Но вы ведь не хотите, чтобы произошел отказ всей программы (а вместе с ней и авария самолета). Скорее вы предпочтете, чтобы программа узнала, что при выполнении данной функции произошел отказ. Поскольку выход за границы диапазона может оказаться временной ситуацией для программы, обрабатывающей данные в режиме реального времени, то программа могла бы отреагировать на такую ошибку просто путем ожидания (простоя) в течение нескольких тактовых циклов, а затем попробовать повторно выполнить свою работу.

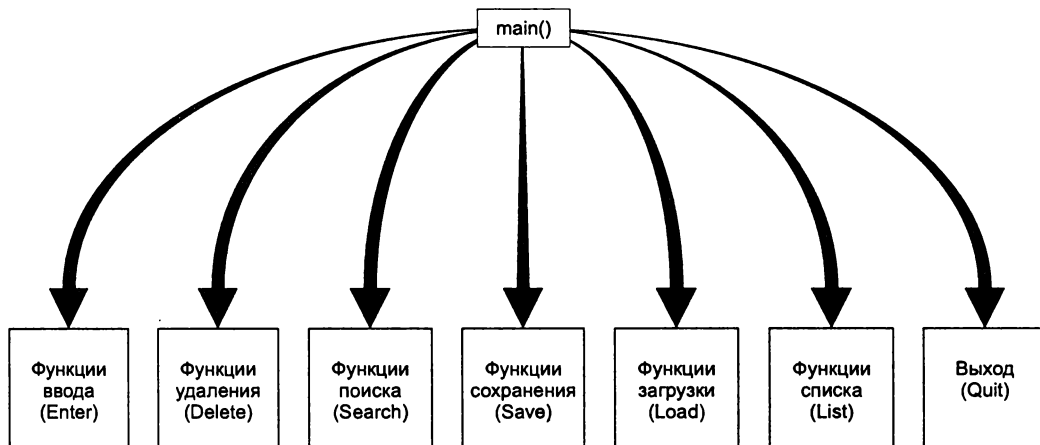


Рис. 27.1. Каждый функциональный модуль имеет только одну точку входа

Имейте в виду, что неукоснительное соблюдение этих правил может оказаться невозможным в любой ситуации, но необходимо придерживаться этих принципов везде, где это только возможно. Подобный подход преследует цель максимизировать в соз-

даваемой программе вероятность восстановления после сбойной ситуации, т.е. чтобы программа работала так, как если бы состояние ошибки не возникало.

На заметку

Читателям, проявляющим повышенный интерес к концепциям построения "пуленепробиваемых" функций, обязательно надо детально исследовать и поэкспериментировать с C++; он обеспечивает значительно более сильный механизм защиты, называемый инкапсуляцией, который еще больше уменьшает вероятность повреждения одной функции другой.

Использование программы MAKE

При создании больших программ разработчиков подстерегает ошибка иного рода, которая проявляется в основном на стадии разработки, но может завести такой проект почти в глухой тупик. Такая ошибка возникает в тех случаях, когда при компиляции и компоновке программы один или несколько файлов ресурсов оказываются устаревшими по сравнению с соответствующими объектными файлами. Если такое случится, то полученная в результате исполняемая программа не будет функционировать так, как предусмотрено в последней реализации исходного кода. Наверно каждый, кто когда-либо участвовал в создании или сопровождении большого программного проекта, сталкивался с подобной проблемой. Чтобы помочь избежать обескураживающих ошибок подобного типа, большинство компиляторов C имеют в своем составе утилиту под названием MAKE, которая помогает синхронизировать файлы ресурсов и объектные файлы. (Точное название MAKE-утилиты, соответствующей вашему компилятору, может немного отличаться от MAKE, поэтому для большей уверенности проверьте себя по документации, прилагаемой к компилятору.)

Программа MAKE автоматизирует процесс перекомпиляции крупных программ, скомпонованных из нескольких файлов. Очень часто в процессе разработки программы во многие файлы вносятся масса незначительных изменений. После этого программа повторно компилируется и тестируется. К сожалению, довольно легко забыть, какие именно файлы нуждаются в перекомпиляции. В такой ситуации вы можете или перекомпилировать все файлы и потерять массу времени, или случайно пропустить файл, который обязательно необходимо перекомпилировать. А это, в свою очередь, может повлечь за собой необходимость в дополнительной, иногда многочасовой, отладке. Программа MAKE решает эту проблему посредством автоматической перекомпиляции только тех файлов, которые претерпели изменения.

Примеры, приведенные в этом разделе, совместимы с программами MAKE, поставляемыми вместе с Microsoft C/C++. На сегодняшний день, Microsoft-версия программы MAKE носит название NMAKE. Эти примеры будут также работать со многими другими широко распространенными MAKE-утилитами, а общие концепции, описанные здесь, применимы для всех MAKE-программ.

На заметку

В последние годы программы MAKE стали очень изощренными. Примеры, приведенные здесь, иллюстрируют только основные возможности MAKE. Стоит подробнее изучить утилиту MAKE, поддерживаемую вашим компилятором. Она может содержать такие функциональные особенности, которые окажутся исключительно полезными в вашей среде разработки.

В своей работе утилита MAKE руководствуется сборочным файлом проекта или так называемым *make-файлом*, который содержит перечень выходных файлов (target files), зависимых файлов (dependent files) и команд. Для генерации *выходного файла* необходимо наличие *файлов, от которых он зависит*. Например, от T.C

зависит файл Т.OBJ, поскольку Т.С необходим для создания Т.OBJ. В процессе работы утилиты MAKE производится сравнение даты выходного файла с датой файла, от которого он зависит. (В данном случае под термином “дата” подразумевается и календарная дата, и время.) Если выходной файл старше, т.е. имеет более позднюю дату создания, чем файл, от которого он зависит (или если выходной файл вообще отсутствует), выполняется указанная последовательность команд. Если эта последовательность команд использует выходные файлы, при построении которых используются файлы, от которых они зависят, то при необходимости модифицируются также и эти используемые файлы¹. Когда процесс MAKE завершится, все выходные файлы будут обновлены. Следовательно, в правильно построенном сборочном файле проекта все исходные файлы, которые требуют компиляции, автоматически компилируются и компоуются, образуя новый исполняемый модуль. Таким образом, данная утилита следит, чтобы все изменения в исходных файлах были отражены в соответствующих им объектных файлах.

В общем виде make-файл выглядит следующим образом:

```
target_file1: dependent_file list
             command_sequence

target_file2: dependent_file list
             command_sequence

target_file3: dependent_file list
             command_sequence

.
.
.
target_fileN: dependent_file list
             command_sequence
```

Имя выходного файла должно начинаться в крайней левой позиции; за ним должно следовать двоеточие и список файлов, от которых он зависит. Последовательность команд, соответствующая каждому выходному файлу, должна предваряться как минимум одним пробелом или одним знаком табуляции. Перед комментариями должен стоять знак #, а сами комментарии могут следовать за списком зависимых файлов и/или последовательностью команд. Кроме того, их можно написать в отдельных строках. Спецификации выходных файлов должны отделяться друг от друга по крайней мере одной пустой строкой.

Самое главное при работе с make-файлом —учитывать следующую особенность: выполнение make-файла заканчивается сразу после того, как удастся обработать первую же цепочку зависимостей. Это означает, что необходимо разрабатывать свои make-файлы таким образом, чтобы зависимости составляли иерархическую структуру. Запомните, что ни одна зависимость не будет считаться успешно обработанной до тех пор, пока все подчиненные ей зависимости (т.е. зависимости более низкого уровня) не будут разрешены.

Чтобы лучше понять, как работает утилита MAKE, давайте рассмотрим очень простую программу. Она состоит из четырех файлов под названием TEST.H, TEST.C, TEST2.C и TEST3.C. Рис. 27.2 иллюстрирует данную ситуацию. (Чтобы лучше понять, о чем идет речь, введите каждую часть программы в указанные файлы.)

¹ Все происходит как при вычислении сложной функции: сначала вычисляются аргументы, от которых она зависит, а если они, в свою очередь, являются сложными функциями, то при необходимости сначала вычисляются их аргументы и т.д. — *Прим. ред.*

TEST.H:

```
extern int count;
```

TEST.C:

```
#include <stdio.h>
void test2(void), test3(void);

int count = 0;

int main(void)
{
    printf("count = %d\n", count);
    test2( );
    printf("count = %d\n", count);
    test3( );
    printf("count = %d\n", count);

    return 0;
}
```

TEST2.C:

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

void test2(void)
{
    count = 30;
}
```

TEST3.C:

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

void test3(void)
{
    count = -100;
}
```

Рис. 27.2. Простая программа, состоящая из четырех файлов

Если в своей работе вы используете Visual C++, следующий make-файл перекомпилирует данную программу после того, как вы внесете в них какие-нибудь изменения:

```
test.exe: test.h test.obj test2.obj test3.obj
    cl test.obj test2.obj test3.obj

test.obj: test.c test.h
    cl -c test.c

test2.obj: test2.c test.h
    cl -c test2.c
```

```
test3.obj: test3.c test.h
cl -c test3.c
```

По умолчанию программа MAKE выполняет директивы, содержащиеся в файле под названием MAKEFILE. Однако, как правило, разработчики предпочитают применять другие имена для своих сборочных файлов проекта. Задать другое имя make-файла можно с помощью опции `-f` в командной строке. Например, если упоминавшийся ранее make-файл называется TEST, то, чтобы с помощью программы NMAKE фирмы Microsoft скомпилировать необходимые модули и создать исполняемый модуль, в командной строке следует набрать нечто подобное следующей строке:

```
nmake -f test
```

(Эта команда подходит для программы NMAKE фирмы Microsoft. Если вы пользуетесь другой утилитой MAKE, возможно, придется использовать другое имя опции.)

Очень большое значение в сборочном файле проекта имеет очередность спецификаций, поскольку, как уже упоминалось ранее, MAKE прекращает выполнение содержащихся в файле директив сразу после того, как она полностью обработает первую зависимость¹. Например, допустим, что ранее упоминавшийся make-файл был изменен, так что он стал выглядеть следующим образом:

```
# Это неправильный make-файл.
test.obj: test.c test.h
cl -c test.c

test2.obj: test2.c test.h
cl -c test2.c

test3.obj: test3.c test.h
cl -c test3.c

test.exe: test.h test.obj test2.obj test3.obj
cl test.obj test2.obj test3.obj
```

Теперь работа будет выполнена неправильно, если файл TEST.H (или любой другой исходный файл) будет изменен. Это происходит потому, что последняя директива (которая создает новый TEST.EXE) больше не будет выполняться.

Использование макросов в MAKE

MAKE позволяет определять макросы в make-файле. Имена этих макросов являются просто метками-заполнителями информации, которая в действительности будет определена или в командной строке, или в макроопределении из make-файла. Общая форма определения макроса следующая:

имя_макроса=определение

Если в макроопределении используется символ пробела, то такое определение следует заключить в двойные кавычки.

После определения макроса его можно использовать в make-файле следующим образом:

\$(имя_макроса)

¹ Т.е. как только построит первый зависимый файл. — Прим. ред.

Вместо каждого вхождения такого оператора подставляется его макроопределение. Например, в следующем make-файле макрос LIBFIL позволяет указать редактору связей библиотеку:

```
LIBFIL = graphics.lib
```

```
prog.exe: prog.obj prog2.obj prog3.obj  
cl prog.obj prog2.obj prog3.obj $(LIBFIL)
```

Многие MAKE-программы имеют дополнительные функциональные возможности, поэтому очень важно внимательно познакомиться с документацией, поставляемой вместе с компилятором.

Применение интегрированной среды разработки

Большинство современных компиляторов поставляются в двух различных видах. Первый вид представляет собой автономный функционально-законченный компилятор, работающий в режиме командной строки. При работе с таким компилятором вначале программист с помощью отдельного редактора создает исходный текст программ. Затем он компилирует свою программу, и, наконец, выполняет ее. Все эти действия выполняются как отдельные команды, вводимые программистом в командной строке. Любая отладка или контроль исходных файлов (например, с помощью утилиты MAKE) также выполняются обособленно. Компилятор, работающий в режиме командной строки, является традиционной формой поставки компиляторов.

Второй вид компиляторов входит в состав интегрированной среды разработки (IDE — integrated development environment), как например, интегрированная среда разработки Visual C++. Выполненный в таком виде компилятор интегрирован вместе с редактором, отладчиком и диспетчером (или менеджером) проектов (который заменяет самостоятельную утилиту MAKE), а также системой поддержки исполнения программ. С помощью интегрированной среды разработки программист может редактировать, компилировать и прогонять программы, не покидая интегрированной среды разработки. Когда впервые была выпущена интегрированная среда разработки, она представляла собой такого неуклюжего и громоздкого монстра, работать с которым было весьма утомительно. Тем не менее, сегодня интегрированные среды разработки, выпускаемые основными производителями компиляторов, могут предложить программистам очень широкие возможности. Если вы не поленитесь заняться установкой параметров интегрированной среды разработки, чтобы они оптимально соответствовали вашим потребностям, то обнаружите, что применение именно такой интегрированной среды значительно упрощает процесс разработки.

Естественно, применяете ли вы интегрированную среду разработки или традиционно набираете все команды в командной строке — это дело вашего личного вкуса. Если вам нравится набирать все команды в командной строке — это тоже будет правильно. К тому же, одно из достоинств этого традиционного подхода заключается в том, что вы лично сами можете выбирать для работы любое инструментальное средство, а не довольствоваться тем, что предложит вам интегрированная среда разработки.

Полный
справочник по



Глава 28

**Производительность,
переносимость и отладка**

Умение писать программы, которые эффективно используют ресурсы системы, легко переносится в другую среду и в которых отсутствуют ошибки, — вот отличительный признак профессионального программиста. И кстати, это именно те вопросы, при решении которых информатика, как научная дисциплина, превращается в “искусство программирования”. В этой главе мы рассмотрим некоторые методы, которые помогают наделить программы столь желанными свойствами.

Эффективность

В данном разделе рассматривается несколько методов, которые помогут повысить эффективность разрабатываемых программ. В программировании под термином “эффективность”, как правило, подразумевается скорость выполнения программы, а также показатели использования ресурсов системы, или и то, и другое вместе. Понятие ресурсов системы охватывает такие параметры, как объем памяти, дискового пространства, процессорное время и тому подобное — в основном все, что можно распределять и расходовать. Ответ на вопрос, является ли некоторая программа эффективной или нет, иногда носит субъективный характер, ведь суждения об эффективности могут меняться в зависимости от конкретной ситуации. Например, методы программирования, которые использовались при создании программы, ориентированной на работу с пользователем (к примеру, текстового процессора), могут совершенно не подходить для фрагмента системного кода, например, сетевого маршрутизатора.

Эффективность часто предполагает компромиссы. Например, стремление заставить программу выполняться быстрее часто приводит к увеличению ее размеров. Такая взаимосвязь особенно сильно проявляется в тех случаях, когда используется линейный код с целью исключения накладных расходов на вызов функций. С другой стороны, желание уменьшить размер программы за счет замены линейного кода вызовами функций иногда приводит к замедлению выполнения программы. Аналогичная ситуация складывается и в отношении эффективного использования дискового пространства. Достижение этой цели часто подразумевает более компактное представление данных, которое может замедлить доступ к данным, из-за накладных расходов, вызванных дополнительной обработкой процессором таких данных. Такие и подобные им компромиссы эффективности могут вызвать чувство полного разочарования и неудовлетворенности, особенно среди неспециалистов и конечных пользователей, которые не могут понять, почему одно влияет на другое. К счастью, существует несколько методов, которые могут одновременно и ускорить выполнение программы, и уменьшить ее размер.

Язык программирования C позволяет эффективно оптимизировать быстродействие или размер программы. В следующих разделах рассказано о нескольких технических приемах оптимизации, уже получивших весьма широкое распространение, хотя творческий, инициативный программист, без сомнения, обязательно найдет новые решения.

Операции увеличения и уменьшения

Уже стало традицией при обсуждении эффективности использования языка C почти всегда начинать с операторов увеличения (инкремента) и уменьшения (декремента). В некоторых ситуациях применение этих операторов помогает компилятору сгенерировать более эффективный объектный код. Рассмотрим, например, следующую последовательность операторов:

```
/* первый метод */
a = a + b ;
b = b + 1 ;

/* второй метод */
a = a + b++ ;
```

В обоих вариантах приведенного примера программы переменной *a* присваивается значение суммы *a+b*, а затем значение *b* увеличивается на единицу. Однако зачастую второй метод приводит к более эффективной программе, поскольку компилятор имеет возможность избежать выполнения дополнительных (а значит, избыточных) инструкций загрузки и записи в память при обращении к переменной *b*. Другими словами, *b* не надо будет загружать в регистр процессора дважды — в первый раз при суммировании с переменной *a*, а второй раз при ее инкрементировании. Хотя некоторые компиляторы будут автоматически оптимизировать оба варианта, и сгенерируют одинаковый объектный код, это не может считаться само собой разумеющимся для всех остальных случаев. В общем случае, аккуратное использование операторов *++* и *--* может увеличить скорость выполнения программы, и в то же время уменьшить ее размер. Поэтому, старайтесь находить именно такие решения, в которых используются эти операторы.

Применение регистровых переменных

Один из наиболее эффективных методов повышения скорости выполнения программного кода состоит в применении к переменным спецификатора класса памяти *register*. Хотя применение регистровых переменных эффективно и в других случаях, они исключительно хорошо подходят для управления циклом. Ведь регистровые переменные хранятся так, что для обращения к ним требуется минимальное время. Целочисленные переменные, к которым применен спецификатор класса памяти *register*, хранятся, как правило, в регистре центрального процессора. Это имеет огромное значение, поскольку скорость выполнения критически важных циклов программы, часто определяет итоговую скорость ее выполнения. Например:

```
for(i=0; i<MAX; i++) {  
    /* что-нибудь выполнить */  
}
```

В этом фрагменте осуществляется многократная проверка и установка нового значения переменной *i*. В частности, всякий раз, когда выполняется цикл, значение *i* проверяется, и если оно не достигло конечного значения, то инкрементируется. Поскольку этот процесс повторяется много раз, время обращения к *i* оказывает существенное влияние на скорость выполнения всего цикла.

Однако не только управляющие переменные циклов, но любые другие переменные, используемые внутри тела цикла, могут заслуживать модификации с помощью спецификатора класса памяти *register*. Например:

```
for(i=0; i<MAX; i++) {  
    sum = a + b;  
    /* ... */  
}
```

В данном примере при каждом повторном выполнении тела цикла осуществляется обращение к переменным *sum*, *a* и *b*. К тому же, при каждой итерации цикла переменной *sum* присваивается новое значение. Таким образом, время, требуемое для обращения к этим переменным, также влияет на общую производительность приведенного цикла.

Спецификатор класса памяти *register* можно применить к любому количеству переменных. Однако ограничения, обусловленные архитектурой центрального процессора, приведут к тому, что в пределах одной функции большинство компиляторов смогут оптимизировать время доступа всего лишь для нескольких переменных. В общем случае всегда можно рассчитывать на то, что в любой момент времени в регистрах центрального процессора обязательно найдется место, по крайней мере, для двух целочисленных переменных. Дополнительно можно использовать и другие виды быстрой памяти, например кэш-память, но возможности ее применения также ограни-

чены. Поскольку применить оптимизацию к каждой переменной, модифицированной с помощью спецификатора класса памяти `register`, скорее всего, будет невозможно, язык C позволяет компилятору игнорировать спецификатор класса памяти `register` и просто разрешает использовать переменную обычным способом. К тому же это правило позволяет код, созданный для одной среды, скомпилировать для другой среды, в которой емкость запоминающего устройства с быстрой выборкой меньше. Поскольку объем запоминающего устройства с быстрой выборкой всегда ограничен, лучше тщательно отбирать только те переменные, реализация быстрого доступа к которым в максимальной степени оптимизирует программу.

Указатели вместо индексации массива

В некоторых случаях индексацию массива целесообразно заменить арифметическими операциями над указателями (например, приращением). Такая замена может привести к уменьшению размера кода и повышению его быстродействия. Например, рассмотрим следующие два фрагмента кода, которые выполняют одну и ту же работу:

Индексация массива

```
for(;;) {  
    a=array[t++];  
    .  
    .  
    .  
}
```

Приращение указателя

```
p=array;  
for(;;) {  
    a=* (p++);  
    .  
    .  
    .  
}
```

Преимущество метода указателей состоит в следующем. Вначале указателю `p` присваивается адрес переменной `array`, тогда для обработки очередного элемента массива необходимо выполнять всего лишь операцию приращения указателя при каждой последующей итерации цикла. Как бы то ни было, при индексации массива всегда нужно вычислять индекс элемента массива, используя значение `t`, а это — более сложная и трудоемкая задача.

Будьте внимательны. В случаях, когда индекс массива вычисляется с помощью сложной формулы, или когда действия над указателями “затеняют” смысл программы, необходимо использовать индексацию массива. Как правило, лучше слегка снизить производительность работы программы, чем пожертвовать ее четкостью и ясностью. К тому же, разница в производительности между индексацией массива и вычислением указателя может оказаться незначительной в случае использования оптимизирующих компиляторов или в случае, если программа предназначена для работы в различных средах или с различными процессорами.

Применение функций

Помните в любой ситуации, что применение автономных функций вместе с локальными переменными помогает формировать фундамент для структурированного программирования. Функции являются строительными блоками в C-программах и одними из самых сильных сторон и главных достоинств C. И только исключительно с этой позиции (и никак иначе!) необходимо рассматривать дальнейший материал этого раздела. Сделав это строгое предупреждение, можно обратить внимание на некоторые, часто используемые при оптимизации программ, особенности C-функций и их разновидности, связанные со скоростью и размером программного кода.

При компиляции функции для хранения передаваемых функции параметров (если они, конечно, имеются), а также любых локальных переменных, используемых этой функцией, компилятор C использует стек. А когда происходит вызов функции, то в

стек помещается еще и адрес возврата в вызывающую программу. (Это позволяет продолжить выполнение подпрограммы с того места, из которого была вызвана функция. Точнее, с команды, следующей за вызовом функции.) Когда функция возвращает управление, этот адрес и все локальные переменные и параметры будут удалены из стека. Процесс заталкивания (записи в стек) этой информации обычно называют *вызывающей последовательностью* (*calling sequence*), а процесс выталкивания данных из стека называется *возвращающей последовательностью* (*returning sequence*)¹. Эти последовательности выполняются в течение определенного промежутка времени, иногда довольно значительного.

Чтобы лучше понять, каким образом вызов функции может замедлить программу, рассмотрим приведенные ниже два фрагмента программного кода:

Вариант 1

```
for(x=1; x<100; ++x) {
    t=compute(x);
}

double compute(int q)
{
    return fabs(sin (q)/100/3.1416);
}
```

Вариант 2

```
for(x=1; x<100; ++x) {
    t=fabs(sin(x)/100/3.1416);
}
```

Хотя в каждом из циклов осуществляется одна и та же операция, Вариант 2 выполняется быстрее, потому что благодаря применению линейного кода были исключены накладные расходы на выполнение вызывающей и возвращающей последовательностей. (Другими словами, код, реализующий функцию `compute()`, просто дублируется внутри цикла, а не вызывается.)

Чтобы лучше понять, в чем состоят непроизводительные издержки, связанные с вызовом функций, давайте рассмотрим ассемблерный код инструкций, которые необходимы для вызова и возврата из функции. Как вам, наверное, известно, многие С-компиляторы имеют специальную опцию для создания файла с ассемблерным кодом; при использовании этой опции создается ассемблерный, а не объектный код. Применение этой опции позволяет исследовать код, созданный компилятором. В последующем примере мы исследуем файл с ассемблерным кодом, созданным с помощью Visual C++ при включенной опции `-Fa`. Мы внимательно рассмотрим этот файл, чтобы воочию убедиться в том, какой код генерируется для вызывающих и возвращающих последовательностей. Возьмем следующую программу:

```
int max(int a, int b);

int main(void)
{
    int x;

    x=max(10, 20);

    return 0;
}

int max(int a, int b)
{
    return a>b ? a : b;
}
```

¹ Имеются в виду, конечно, последовательности команд. — Прим. ред.

Для нее получится следующий ассемблерный код. Вызывающие и возвращающие последовательности отмечены в листинге комментариями, начинающимися со знака звездочка (*); эти комментарии добавлены автором книги. Как вы можете убедиться, вызывающие и возвращающие последовательности занимают по объему довольно значительную часть программного кода.

```

        TITLE      test.c
        .386P
include listing.inc
if @Version gt 510
.model FLAT
else
_TEXT   SEGMENT PARA USE32 PUBLIC 'CODE'
_TEXT   ENDS
_DATA   SEGMENT DWORD USE32 PUBLIC 'DATA'
_DATA   ENDS
_CONST  SEGMENT DWORD USE32 PUBLIC 'CONST'
_CONST  ENDS
_BSS    SEGMENT DWORD USE32 PUBLIC 'BSS'
_BSS    ENDS
_TLS    SEGMENT DWORD USE32 PUBLIC 'TLS'
_TLS    ENDS
FLAT    GROUP _DATA, CONST, _BSS
        ASSUME    CS: FLAT, DS: FLAT, SS: FLAT
endif
PUBLIC  _max
PUBLIC  _main
_TEXT   SEGMENT
_x$ = -4
_main   PROC NEAR
; File ex2.c
; Line 4
        push     ebp
        mov      ebp, esp
        push     ecx
; Line 7
; *****
; Это начало вызывающей последовательности.
; *****
        push     20 ; 00000014H
        push     10 ; 0000000aH
        call     _max
; *****
;
; *****
; Следующая строка относится к возвращающей последовательности.
; *****
        add      esp, 8
        mov      DWORD PTR _x$[ebp], eax
; Line 9
        xor      eax, eax
; Line 10
        mov      esp, ebp
        pop      ebp
        ret      0
_main   ENDP
_a$ = 8
_b$ = 12

```

```

_max      PROC NEAR
; Line 13
; *****
; Дополнительный фрагмент вызывающей последовательности.
; *****
        push     ebp
        mov      ebp, esp
        push     ecx
; *****
; Line 14
        mov      eax, DWORD PTR _a$[ebp]
        cmp      eax, DWORD PTR _b$[ebp]
        jle      SHORT $L48
        mov      ecx, DWORD PTR _a$[ebp]
        mov      DWORD PTR -4+[ebp], ecx
        jmp      SHORT $L49
$L48:
        mov      edx, DWORD PTR _b$[ebp]
        mov      DWORD PTR -4+[ebp], edx
$L49:
; *****
; Возвращающая последовательность.
; *****
        mov      eax, DWORD PTR -4+[ebp]
; Line 15
        mov      esp, ebp
        pop      ebp
        ret      0
_max      ENDP
_TEXT     ENDS
END

```

Реально получаемый код зависит от особенностей реализации конкретного компилятора и типа используемого процессора, но в основном он будет похож на приведенный выше листинг.

Однако из приведенных рассуждений совсем не следует, что с целью сокращения времени выполнения в программах нужно использовать всего лишь несколько, но зато громадных функций. Ведь это далеко не самый лучший стиль программирования. Во-первых, в абсолютном большинстве случаев незначительный выигрыш во времени, получаемый за счет отказа от вызова функций, не имеет столь большого значения по сравнению с весьма ощутимой потерей структурированности. К тому же, только этой проблемой дело не ограничивается. Замена функций, которые используются несколькими подпрограммами, линейным кодом приводит к непомерному “разрастанию” программы, поскольку одинаковый код повторяется несколько раз. Не забывайте, что подпрограммы были созданы отчасти для того, чтобы более эффективно использовать память. Фактически именно поэтому считается, что уменьшение времени выполнения программы ведет к увеличению ее размеров. И наоборот, уменьшение размера программы обычно приводит к замедлению скорости ее выполнения.

Чтобы применить линейный код как метод ускорения программы, в компиляторе, совместимом с C99, предусмотрено зарезервированное слово `inline` для создания встраиваемых функций (при компиляции вызов такой функции заменяется ее кодом). Используя это слово, вам не придется вручную каждый раз дублировать исходный код каждой функции. Если же ваш компилятор со стандартом C99, то чтобы получить аналогичный результат, следует использовать макросы с параметрами в тех местах, где имеется такая возможность. Естественно, макросы с параметрами не предоставляют той гибкости, которой обладают функции, в описании которых применяется слово `inline`.

Перенос программ

Очень часто (и это стало обыденной практикой) программы, написанные для одной машины, необходимо перенести на другой компьютер, оснащенный другим процессором или другой операционной системой, а зачастую и тем и другим одновременно. Этот процесс носит название *перенос*, или *перенесение* (*porting*), и в одних случаях он может оказаться очень простым, а в других — предельно трудным. Это зависит от того, каким образом была первоначально написана программа. Поэтому, программа, которая легко поддается переносу, называется *переносимой*, *мобильной*, или *машинонезависимой*¹ (*portable*). Если программа не относится к разряду переносимых, как правило, это объясняется тем, что она содержит большое количество элементов, *зависящих от типа машины*, — то есть, она имеет фрагменты кода, которые будут выполняться только в одной определенной операционной системе или на одном конкретном процессоре. Язык С позволяет создавать переносимый код, но для достижения этой цели необходимо проявлять особую тщательность и внимание к деталям. В данном разделе рассматриваются несколько конкретных проблем, возникающих при создании машинонезависимых программ и предлагается ряд решений таких проблем.

Использование #define

Возможно, самый простой и эффективный способ сделать программы переносимыми состоит в том, чтобы представить каждое зависящее от типа системы или процессора “магическое число” макросом `#define`. Магическими эти числа были названы потому, что они представляют собой такие параметры системы, как размер буфера, используемого для обращения к диску, специальные команды управления экраном и клавиатурой, а также информацию о распределении памяти, иными словами, все то, что может измениться при переносе программы. Такие `#define`-определения не только делают все магические числа очевидными для программиста, выполняющего перенос, но к тому же упрощают выполнение всей работы. Ведь поскольку их значения необходимо будет изменить только однажды и в одном месте, следовательно, не придется “перетряхивать” всю программу.

Например, рассмотрим оператор `fread()`, который по своей природе является непереносимым:

```
■ fread(buf, 128, 1, fp);
```

В данном случае проблема заключается в том, что размер буфера (число 128) является жестко запрограммированным параметром функции `fread()`. Это значение может вполне подходить для одной операционной системы, но окажется меньше оптимальной величины для другой. А вот более удачный способ кодирования этой же функции:

```
■ #define BUF_SIZE 128  
fread(buf, BUF_SIZE, 1, fp);
```

В последнем варианте при переносе на другую систему понадобится всего лишь изменить определение `#define` — и все ссылки на `BUF_SIZE` будут исправлены автоматически. Это не только облегчает процесс внесения изменений, но и вдобавок уберегает от массы ошибок при редактировании. Помните, что в реальной программе может присутствовать бездна ссылок на `BUF_SIZE`, поэтому переносимость программы возрастает многократно.

¹ Термины *портability* и *портативный* к настоящему времени несколько утратили свою первоначальную популярность. — Прим. ред.

Зависимость от операционной системы

Код практически всех коммерческих программ адаптирован для конкретной операционной системы, под управлением которой предполагается их работа. К примеру, программа, написанная для Windows 2000, может использовать многопоточковую многозадачность, тогда как программа, написанная для 16-разрядной Windows 3.1, не может. Дело в том, что определенная привязка к особенностям конкретной операционной системы совершенно необходима, чтобы получить по-настоящему хорошие, быстродействующие и по-коммерчески жизнеспособные программы. Однако с другой стороны, зависимость от операционной системы усложняет процесс переноса программ.

Хотя и не существует жестких правил, следуя которым можно было бы минимизировать зависимость разрабатываемых программ от типа операционной системы, позвольте предложить маленький совет. Отделяйте в разрабатываемой программе те части, которые относятся непосредственно к приложению от тех фрагментов, которые осуществляют взаимодействие с операционной системой. Тогда при переносе программы в новую среду потребуется изменить только модули интерфейса.

Различия в размерах данных

Если вы хотите написать переносимый код, никогда не следует полагаться на ожидаемые размеры данных. Например, надо всегда учитывать отличия между 16-разрядной и 32-разрядной средами. Размер слова в 16-разрядном процессоре равен 16 битам, а в 32-разрядном процессоре — 32 битам. Поскольку размер слова часто совпадает с размером данных типа `int`, то код, созданный в предположении, что переменные типа `int` являются, к примеру, 16-разрядными, не будет корректно работать после переноса его в 32-разрядную среду. Чтобы избежать жесткой привязки к размеру, там, где программе понадобятся сведения о количестве байтов, составляющих какую-нибудь величину, обязательно используйте оператор `sizeof`. Например, следующее выражение заносит значение типа `int` в дисковый файл и будет работать в любой среде:

```
fwrite(&i, sizeof(int), 1, stream);
```

Отладка

Перефразируя Томаса Эдисона, можно утверждать, что программирование на 10 процентов состоит из вдохновения и на 90 процентов из отладки. Все действительно квалифицированные программисты являются хорошими отладчиками. Чтобы научиться предотвращать множество ошибок, целесообразно рассмотреть некоторые довольно распространенные действия, которые могут привести к их появлению.

Ошибки очередности вычисления

В большинстве С-программ применяются операторы инкрементирования и декрементирования, а порядок следования этих операторов, как вы помните, имеет большое значение в зависимости от того, предшествуют они или следуют за переменной. Рассмотрим следующий случай:

```
y=10;          y=10;  
x=y++;          x=++y;
```

Приведенные две последовательности не эквивалентны. Та, что слева, присваивает переменной `x` значение 10, а затем инкрементирует `y`. В другой же последовательности (справа) `y` сначала инкрементируется, и в результате этого становится равным 11, и только

затем значение 11 присваивается переменной *x*. Таким образом, в первом случае *x* равно 10, а во втором случае *x* — 11. В общем случае в сложных выражениях префиксная операция инкрементирования (или декрементирования) осуществляется перед вычислением значения операнда, используемого в последующих действиях. Постфиксный инкремент (или декремент) выполняется в сложных выражениях после того, как значение операнда вычислено. Если забыть об этих правилах, проблем не миновать.

Путь, обычно ведущий к возникновению ошибки очередности вычисления, заключается в изменении имеющейся последовательности операторов. Например, при оптимизации фрагмента кода вы могли бы изменить следующую последовательность:

```
/* первоначальный код */
x = a + b;
a = a + 1;
```

и представить ее в таком виде:

```
/* "усовершенствованный" код - ошибка! */
x = ++a + b;
```

Проблема заключается в том, что эти два фрагмента кода не дают одинаковый результат. Причина состоит в том, что второй способ инкрементирует переменную *a* до того, как она суммируется с *b*. А такое действие в первоначальном варианте не было предусмотрено!

Подобные ошибки относятся к разряду трудно обнаруживаемых. Могут быть ключи-подсказки, например циклы, выполняющиеся неправильно, или процедуры, которые не работают из-за таких ошибок. Если у вас возникает сомнение в правильности оператора, перекодируйте его таким образом, чтобы быть уверенным в нем на все 100 процентов.

Проблемы с указателями

Очень распространенной ошибкой в С-программах является неправильное применение указателей. Проблемы с указателями условно можно разделить на две основные категории: неправильное представление об использовании косвенной адресации и об операциях над указателями вообще, а также случайное (точнее непредумышленное) применение недействительных или неинициализированных указателей. Решить первую проблему несложно: просто разберитесь окончательно и до конца в том, что означают операторы *** и *&*! Справиться со второй категорией проблем с указателями несколько сложнее.

Ниже приведена программа, иллюстрирующая оба типа ошибок, связанных с указателями:

```
/* Эта программа содержит ошибку. */
#include <stdlib.h>
#include <stdio.h>

int main(void)
{
    char *p;

    *p = (char *) malloc(100); /* эта строка содержит ошибку */
    gets(p);
    printf(p);

    return 0;
}
```

При запуске такой программы скорее всего произойдет ее сбой. Объясняется это тем, что значение адреса, возвращаемого функцией *malloc()*, не было присвоено указателю *p*, а было размещено в ячейку памяти, на которую указывает *p*, адрес кото-

рой в данном случае неизвестен (и в общем случае, непредсказуем). Данный тип ошибки представляет пример фундаментального непонимания выполнения операторов над указателями (а именно выполнения операции *). Как правило, такая ошибка в программах на С допускается начинающими программистами, но иногда эта нелепая оплошность встречается и у опытных профессионалов! Чтобы исправить эту программу, необходимо заменить строку с ошибкой следующей корректной строкой:

```
■ p = (char *) malloc(100); /* эта строка правильная */
```

Кроме того, данная программа содержит еще одну, причем более коварную ошибку. В ней отсутствует динамическая проверка значения адреса, возвращаемого функцией `malloc()`. Помните, если память будет исчерпана, `malloc()` возвратит значение `NULL`, а тогда указатель использовать нельзя. Использование `NULL` в качестве указателя объекта недопустимо и практически всегда ведет к аварийному завершению программы. Вот исправленный вариант данной программы, в который включена проверка доступности указателя:

```
/* Теперь эта программа написана корректно. */  
  
#include <stdio.h>  
#include <stdlib.h>  
  
int main(void)  
{  
    char *p;  
  
    p = (char *) malloc(100); /* эта строка не содержит ошибок */  
  
    if(!p) {  
        printf("Out of memory - Нет памяти.\n");  
        exit(1);  
    }  
  
    gets(p);  
    printf(p);  
  
    return 0;  
}
```

Следующая часто встречающаяся ошибка заключается в том, что программист забывает инициализировать указатель перед использованием. Обратимся к следующему фрагменту программы:

```
■ int *x;  
  *x=100;
```

Выполнение такого кода обязательно приведет к проблемам, поскольку указатель `x` не был инициализирован, а значит, едва ли можно ожидать, что он указывает туда, куда нужно. Фактически вы не знаете, куда указывает `x`. Присвоение какого-либо значения этой неизвестной области памяти может разрушить что-то, имеющее огромное значение, например другой фрагмент программы или данные.

Самая большая неприятность с “дикими” (т.е. непредсказуемыми) указателями состоит в том, что их невероятно тяжело обнаружить. Если вы присваиваете значение посредством указателя, который не содержит действительный адрес, ваша программа в одних случаях может функционировать вполне корректно, а в других — завершаться аварийным отказом. Чем меньше размер программы, тем выше вероятность, что она будет работать правильно, даже с “блуждающим” указателем. Это объясняется тем, что в таком случае программой используется очень маленький объем памяти, поэтому довольно велики шансы того, что

указатель-нарушитель указывает на неиспользуемую область памяти. Но по мере увеличения объема программы подобные сбои будут происходить все чаще и чаще. Но вы скорее попытаетесь объяснить их последними внесенными в программу дополнениями или изменениями, и вряд ли свяжете с ошибками в использовании указателей. Следовательно, вы будете искать ошибки совершенно не в том месте.

Подсказкой для распознавания проблем с указателем является то, что такие ошибки часто проявляются нерегулярно и зачастую странным образом. Один раз программа работает вполне корректно, а другой раз — неправильно. Иногда некоторые переменные содержат “мусор”, хотя на то нет каких бы то ни было видимых причин. Когда возникают подобные проблемы, проверьте все указатели. Собственно говоря, вы всегда должны проверять все указатели, как только начнут проявляться любые ошибки¹.

Возможно, утешением, станет то, что хотя указатели могут доставить множество хлопот, тем не менее, они являются одним из наиболее мощных средств языка C и стоят преодоления любой проблемы, которую они могут вам преподнести. Просто постарайтесь с самого начала изучить их правильное применение.

Интерпретация синтаксических ошибок

Время от времени вы будете сталкиваться с синтаксическими ошибками, сообщения о которых покажутся вам абсурдными и бессмысленными. То ли сообщение об ошибке зашифровано, то ли ошибка, описание которой приводится в сообщении, вообще не похожа на ошибку. Тем не менее, в большинстве случаев в вопросах обнаружения ошибок компилятор оказывается прав. Просто в подобных случаях сам текст сообщения об ошибке чуть-чуть не дотягивает до совершенства. При поиске причин необычных синтаксических ошибок, как правило, необходимо при чтении программы немного возвратиться назад. Поэтому, если вы столкнулись с сообщением об ошибке, которое, судя по всему, не имеет смысла, попробуйте поискать синтаксическую ошибку одной двумя строками выше по тексту вашей программы.

С одной из особенно сногшибательных ошибок можно познакомиться ближе, если вы попытаетесь скомпилировать следующий код:

```
char *myfunc(void);

int main(void)
{
    /* ... */
}

int myfunc(void) /* сообщение об ошибке указывает сюда */
{
    /* ... */
}
```

¹ Вот еще несколько универсальных советов, как избавиться от проблем: “Всегда все тщательно проверяйте!”, “Никогда не делайте ошибок!”, “Всегда все кодируйте правильно!” и т.д. Когда я слышу подобные советы, я вспоминаю, как всем известная Алиса реагировала на подобные высказывания: “Но ведь я не могла!”, на что Шалтай-Болтай ей отвечал: “Я и не говорю, что ты могла; я говорю, что ты должна была!” Честно говоря, позиция Алисы мне очень и очень близка, ведь недаром же известный философ Хинтика доказал, что аморально требовать от человека то, чего он не может выполнить. Если теперь предположить (вопреки тому, что заказчики считают, что программисты *обязаны быть роботами*), что все программисты — люди, то мы можем прийти к заключению, что не во всех программах всегда проверяется применение всех указателей. (С точки зрения заказчиков это, конечно, недопустимо.) А потому совет, даваемый автором, хотя и смахивает на один из универсальных, вполне заслуживает того, чтобы прислушаться к нему. Статистика же подтверждает, что это — самый лучший совет из всех, которые можно дать при отладке программ с указателями! Так что ничего не поделаешь — не ленитесь! — Прим. ред.

Ваш компилятор выдаст сообщение об ошибке вместе с таким вот разъяснением:

```
Type mismatch in redeclaration of myfunc(void)
(Несоответствие типов при повторном объявлении myfunc(void))
```

Это сообщение относится к строке листинга программы, которая помечена комментарием о наличии ошибки. Как такое возможно? Ведь в этой строке нет двух функций `myfunc()`. А разгадка состоит в том, что прототип в верхней строке программы показывает, что `myfunc()` возвращает значение типа указатель на символ. Это ведет к тому, что в таблице идентификаторов компилятор заполняет строку, содержащую эту информацию. Когда затем в программе компилятор встречает функцию `myfunc()`, то теперь тип результата указывается как `int`. Следовательно, вы “повторно объявили”, другими словами “переопределили” функцию.

Другая синтаксическая ошибка, которую трудно сразу правильно истолковать, генерируется при попытке скомпилировать следующий код:

```
/* В тексте данной программы имеется синтаксическая ошибка */
#include <stdio.h>

void func1(void);

int main(void)
{
    func1();

    return 0;
}

void func1(void);
{
    printf("Это в func1.\n");
}
```

Здесь ошибка состоит в наличии точки с запятой после определения функции `func1()`. Компилятор будет рассматривать это как выражение, находящееся за пределами какой бы то ни было функции, что является ошибкой. Однако различные компиляторы по-разному сообщают об этой ошибке. Некоторые компиляторы выводят в сообщении об ошибке такой текст: `bad declaration syntax` (неправильный синтаксис объявления), и в то же время указывают на первую открытую скобку после функции `func1()`. Поскольку вы привыкли в конце выражений ставить точку с запятой, подобную ошибку очень трудно заметить.

Ошибки, вызванные “потерей” единицы

Как известно, в С нумерация индексов любого массива начинаются с нуля. Тем не менее, даже опытные профессионалы в пылу творческого вдохновения, бывало, забывали это общеизвестное правило! Рассмотрим следующую программу, которая, как предполагается, должна инициализировать массив из ста целых чисел:

```
/* Эта программа работать не будет. */

int main(void)
{
    int x, num[100];

    for(x=1; x <= 100; ++x) num[x] = x;

    return 0;
}
```

Цикл `for` в этой программе выполнен неправильно по двум причинам. Во-первых, он не инициализирует `num[0]`, первый элемент массива `num`. Во-вторых, он пытается проинициализировать элемент массива с номером на единицу больше, чем у последнего элемента массива, поскольку `num[99]` как раз и является последним элементом массива, а параметр цикла достигает значения 100. Правильно было бы записать эту программу следующим образом:

```
/* Здесь все правильно. */  
  
int main(void)  
{  
    int x, num[100];  
  
    for(x=0; x < 100; ++x) num[x] = x;  
  
    return 0;  
}
```

Помните, в массиве из 100 элементов элементы пронумерованы числами от 0 до 99.

Ошибки из-за нарушения границ

И в среде прогона программ, написанных на языке C, и во многих стандартных библиотечных функциях почти не имеется (а иногда они вообще отсутствуют) средств динамической проверки принадлежности к диапазону (т.е. средств контроля границ). Например, если в программе произойдет выход за границы массива, то такая ошибка может остаться незамеченной. Рассмотрим следующую программу, которая должна считывать строку символов из буфера клавиатуры и отображать ее на экране монитора:

```
#include <stdio.h>  
  
int main(void)  
{  
    int var1;  
    char s[10];  
    int var2;  
  
    var1 = 10;    var2 = 10;  
    gets(s);  
    printf("%s %d %d", s, var1, var2);  
  
    return 0;  
}
```

В этом фрагменте нет очевидных ошибок кодирования. Тем не менее, вызов функции `gets()` с параметром `s` может косвенно привести к ошибке. В данной программе переменная `s` объявлена как массив символов (строка длиной в 10 знаков). Но что произойдет, если пользователь введет больше десяти знаков? Это приведет к выходу за границы массива `s`, и значение переменной `var1` или `var2`, а возможно, и их обеих будет перезаписано. Следовательно, `var1` и (или) `var2` не будут содержать правильных значений. Это вызвано тем, что для хранения локальных переменных все C-компиляторы применяют стек. Переменные `var1`, `var2`, а также `s` могут располагаться в памяти так, как показано на рис. 28.1. (Ваш компилятор C может поменять порядок следования переменных `var1`, `var2` и `s`.)

Предположим, что порядок распределения ячеек памяти совпадает с изображенным на рис. 28.1. Тогда, если произойдет выход за границы массива `s`, то дополнительные (лишние) символы будут помещены в область, в которой должна находиться переменная `var2`. Это практически уничтожит информацию, ранее записанную там.

Поэтому на экран будет выведено не число 10 в качестве значения обеих целых переменных, а в качестве значения переменной, поврежденной в результате выхода за границы массива *s*, будет отображено что-нибудь другое. А вы можете искать ошибку совсем в другом месте.

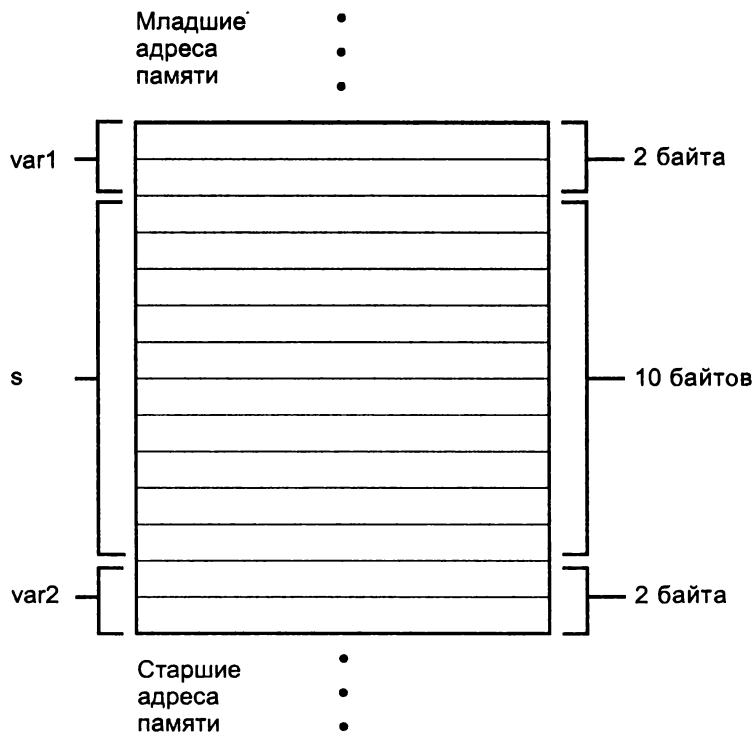


Рис. 28.1. Размещение в памяти переменных var1, var2 и s (исходя из предположения, что на целое число выделяется 2 байта)

В рассмотренной программе потенциальная ошибка из-за выхода за границы может быть исключена за счет применения функции `fgets()` вместо `gets()`. Функция `fgets()` предоставляет возможность устанавливать максимальное количество считываемых символов. Единственная проблема состоит в том, что `fgets()` считывает и сохраняет еще и символ разделителя строк, поэтому в большинстве приложений его необходимо будет удалять.

Пропуск прототипов функций

В современной среде программирования отсутствие даже одного прототипа функции является непростительным упущением, “отступничеством” от мудрых принципов и здравого смысла. Чтобы понять почему именно, рассмотрим следующую программу, которая выполняет умножение двух чисел с плавающей запятой:

```
/* Эта программа содержит ошибку. */
#include <stdio.h>

int main(void)
{
    float x, y;
```



```

scanf("%f%f", &x, &y);
printf("%f", mul(x, y));

return 0;
}

double mul(float a, float b)
{
    return a*b;
}

```

В данном случае, поскольку прототип функции `mul()` отсутствует, при компиляции функции `main()` предполагается, что в результате выполнения `mul()` будет возвращена целочисленная величина. Но в действительности `mul()` возвращает число с плавающей запятой. Допустим, что для целого числа выделяется 4 байта, а для чисел двойной точности (`double`) — 8 байтов. Это значит, что в действительности только четыре байта из восьми, необходимых для двойной точности, будут использованы в операторе `printf()` внутри функции `main()`. Это приведет к неправильному ответу, выводимому на экран дисплея.

Чтобы исправить эту программу, достаточно создать прототип функции `mul()`. Корректный вариант программы будет выглядеть следующим образом:

```

/* Это правильная программа. */
#include <stdio.h>

double mul(float a, float b);

int main(void)
{
    float x, y;

    scanf("%f%f", &x, &y);
    printf("%f", mul(x, y));

    return 0;
}

double mul(float a, float b)
{
    return a*b;
}

```

В данном случае прототип указывает, что при компиляции функции `main()` необходимо учитывать, что функция `mul()` возвращает значение с удвоенной точностью.

Ошибки при задании аргументов

Тип любого формального параметра, должен соответствовать типу фактического параметра. Хотя благодаря прототипам функций компиляторы могут обнаруживать многие несоответствия типов аргументов (параметров), они не могут обнаружить все. Более того, когда функция имеет переменное количество параметров, компилятор не может обнаружить несоответствие их типов. Например, рассмотрим функцию `scanf()`, которая принимает большое количество разнообразных аргументов. Не забывайте, что `scanf()` ожидает принять *адреса* своих аргументов, а не их значения. И никакая сила не сделает за вас правильную подстановку. Например, следующая последовательность операторов

```
int x;  
scanf("%d", x);
```

содержит ошибку, поскольку передается значение переменной *x*, а не ее адрес. Тем не менее, вызов этой функции `scanf()` будет скомпилирован без сообщения об ошибке, и лишь во время выполнения этого оператора выявится ошибка. Правильный вариант вызова функции `scanf()` приведен ниже:

```
scanf("%d", &x);
```

Переполнение стека

Все компиляторы C используют стек для хранения локальных переменных, адресов возврата и передаваемых функциям параметров. Однако стек не безграничен, и, в конце концов, может быть исчерпан. Тогда попытка записи очередного элемента в него приведет к переполнению стека. Когда такое происходит, программа или полностью “умирает”, или продолжает выполняться в ненормальном причудливом стиле. Самое неприятное в переполнении стека заключается в том, что оно в большинстве случаев происходит безо всякого предупреждения и оказывает на программу столь серьезное воздействие, что определить, что именно было сделано неправильно, иногда бывает невероятно трудно. Единственной приемлемой подсказкой может служить то, что в некоторых случаях переполнение стека вызвано выходом из-под контроля рекурсивных функций. Если в вашей программе используются рекурсивные функции и вы столкнулись с необъяснимыми сбоями в ее работе, проверьте условия завершения в рекурсивных функциях.

И еще одно замечание. Некоторые компиляторы позволяют увеличить объем памяти, резервируемой под стек. Если ваша программа во всем остальном не имеет ошибок, но быстро исчерпывает стековое пространство (возможно из-за глубокой степени вложенности или рекурсивности функций), необходимо просто увеличить размер стека.

Применение отладчика

Многие компиляторы поставляются вместе с отладчиком, который представляет собой программу, помогающую отладить разрабатываемый код. В общем случае отладчики позволяют шаг за шагом исполнять код разрабатываемой программы, устанавливать точки останова и контролировать содержимое различных переменных. Современные отладчики, например такие, как поставляемые в составе пакета Visual C++, являются действительно замечательными инструментальными средствами, которые могут оказать существенную помощь в обнаружении ошибок в разрабатываемом коде. Хороший отладчик стоит дополнительного времени и усилий, которые необходимы на его изучение, чтобы в дальнейшем эффективно его применять. Как бы то ни было, хороший программист *никогда* не откажется от работы с отладчиком для реализации надежного проекта и выполнения тонких работ.

Теория отладки в общих чертах

Каждый разработчик имеет свой собственный подход в программировании и отладке. Тем не менее, длительный опыт показывает, что существуют технические приемы, которые значительно лучше, чем остальные. В отношении отладки считается, что наиболее эффективным методом в плане времени и стоимости является инкрементное (нарастающее) тестирование, даже если может показаться, что этот подход может замедлить на первых порах процесс разработки. *Инкрементное тестирование* является технологическим приемом, гарантирующим, что вы всегда будете иметь работоспособную программу. В чем же его суть? Уже на самых ранних стадиях процесса

разработки функциональный блок. *Функциональный блок* — это просто фрагмент работающего кода. По мере добавления нового кода к этому блоку, он тестируется и отлаживается. Таким способом программист может обнаруживать ошибки без особого труда, поскольку, вероятнее всего, ошибки будут присутствовать в более новом коде (добавке) или возникать из-за плохого взаимодействия с функциональным блоком.

Время отладки прямо пропорционально общему количеству строк кода, в котором программист ищет ошибки. Благодаря инкрементному тестированию количество строк кода, в котором необходимо искать ошибки, ограничено как правило, количеством вновь добавленных строк. Другими словами, ошибка, скорее всего, содержится в строках, которые не входят в состав функционального блока. Эта ситуация проиллюстрирована на рис. 28.2. Любому программисту хочется минимизировать объем отлаживаемого фрагмента программы. Метод инкрементного тестирования позволяет не тестировать те участки, где эта работа уже была проведена. Таким образом, можно уменьшить область, в которой вероятнее всего прячется ошибка.

Крупные проекты часто можно разбить на несколько модулей, слабо взаимодействующих между собой. В таких случаях можно выделить несколько функциональных блоков, что позволит вести параллельную разработку проекта.

Инкрементное тестирование — просто технологический прием, благодаря которому который всегда можно иметь работоспособный код. Поэтому всякий раз, когда появляется возможность выполнить кусочек разрабатываемой программы, вы должны запустить его на выполнение и тщательно протестировать его. По мере добавления к программе новых фрагментов продолжайте тестировать их, а также их интерфейс с уже проверенным функциональным кодом. Этот способ позволяет разрабатывать программу так, что большинство ошибок будет сконцентрировано в небольшой области кода. Конечно, вы никогда не должны упускать из виду то, что ошибка могла быть пропущена и в функциональном блоке, но все же данный метод тестирования уменьшает вероятность такого случая.



Рис. 28.2. При регулярном использовании метода инкрементного тестирования ошибки, если они есть, вероятнее всего находятся в добавленном коде

Полный справочник по



Часть VI

Интерпретатор языка C

В заключительной части проектируется интерпретатор языка C, благодаря чему сразу достигаются две важные цели. Во-первых, разработка этого проекта позволяет продемонстрировать несколько аспектов языка C, важных с точки зрения проектирования больших систем. Во-вторых, разработка такого проекта позволяет увидеть природу языка C как бы изнутри и узнать, почему язык имеет именно такую структуру.

Полный
справочник по



Глава 29

Интерпретатор языка C

Работа с интерпретаторами увлекает! А для программиста, пишущего программы на С, нет работы более увлекательной, творческой и интересной, чем работа с интерпретатором С!

Завершающая глава книги посвящена теме, представляющей интерес для всех программистов, работающих с языком С, и кроме того, она иллюстрирует основные средства языка путем создания интерпретатора языка Little С.

Не отрицая всей значимости и актуальности компиляторов, приходится признать, что их создание — очень трудный и длительный процесс. Фактически, одно только создание библиотеки рабочих программ компилятора уже представляет собой чрезвычайную сложную задачу. В то же время создание интерпретатора языка — сравнительно более легкая задача, “выполнимая” в рамках одной главы. Нужно отметить, что легче понять принципы работы хорошо сконструированного интерпретатора, чем аналогичного компилятора. Помимо простоты разработки, интерпретатор языка содержит нечто такое, чего нет в компиляторе, — движущий механизм, фактически выполняющий программу. Необходимо помнить, что *компилятор только транслирует* исходный текст программы, то есть придает программе тот вид, в котором она выполняется компьютером, а *интерпретатор выполняет* программу. Именно это различие делает интерпретаторы более интересной темой для рассмотрения.

Большинство программистов, пишущих программы на С, оценили не только мощност и гибкость этого языка, но и его необычную формальную красоту, сделавшую его особенно привлекательным для специалистов. Благодаря его логичности и чистоте о языке С часто говорят как об элегантном языке.

К настоящему времени об использовании языка С и принципах программирования на нем написано довольно много, однако исследования работы языка “изнутри” встречаются не часто. Поэтому лучшим способом завершения этой книги будет создание программы на С, интерпретирующей подмножество этого же языка.

В данной главе рассматривается разработка интерпретатора, способного выполнять программы, написанные на подмножестве языка С. Этот интерпретатор не только вполне работоспособен, он также написан таким образом, что его легко расширять, добавляя новые средства, отсутствующие даже в стандарте С. Читатель, который не знает, как выполняется программа на языке С, будет приятно удивлен прямолинейностью ее выполнения. Язык С — один из самых теоретически последовательных языков программирования. Ознакомившись с этой главой, читатель не только получит интерпретатор С, пригодный для использования и расширения, но и существенно улучшит свое понимание структуры языка. К тому же, сама работа с этим интерпретатором — весьма интересное и увлекательное занятие.

На заметку

Представленный в этой главе интерпретатор С имеет довольно длинный исходный текст программы, но это не будет проблемой. Прочтя изложенный материал, читатель без труда поймет текст программы и принцип ее работы.



Практическое значение интерпретаторов

Интерпретатор С весьма интересен как объект исследований и экспериментов, кроме того, интерпретаторы вообще имеют немалое практическое значение.

Программы на С обычно компилируются. Главная причина этого в том, что язык С широко используется для создания коммерческого программного продукта. Для этой цели скомпилированная программа считается более предпочтительной потому, что компиляция позволяет сохранить конфиденциальность исходного текста программы, предотвратить изменение этого текста пользователем, эффективно использовать ресурсы компьютера. Кроме названных, существует немало и других причин. Видимо,

компиляторы всегда будут доминировать при разработке программного продукта на основе С. Тем не менее, программа на любом языке может быть как скомпилирована, так и интерпретирована. В последние годы на рынке программных продуктов появилось даже несколько новых интерпретаторов С.

Можно назвать две традиционные причины того, что интерпретаторы продолжают использоваться: их легко сделать интерактивными, а также они очень облегчают отладку программы. Однако в последние годы разработчики компиляторов обычно создают интегрированные среды разработки (Integrated Development Environments, IDEs), в которых предусмотрены средства для интерактивной работы и отладки не хуже, чем имеющиеся у интерпретаторов. Поэтому обе вышеназванные причины применения интерпретаторов сейчас уже не актуальны. Однако интерпретаторы продолжают использоваться. Например, большинство программ, написанных на языках запросов к базам данных, сейчас интерпретируются, а не компилируются. Многие языки управления промышленными роботами также интерпретируются.

В последние годы проявилось еще одно преимущество интерпретаторов: повышенная переносимость на различные инструментальные комплексы. Характерный пример этого — язык Java. С самого начала Java разрабатывался как язык, предназначенный для интерпретации. Сделано было это специально для того, чтобы программы, написанные на нем, можно было выполнять на любом компьютере и в любой среде, содержащей интерпретатор Java. Такое свойство языка является чрезвычайно ценным, если программа предназначена для работы в распределенных сетевых системах наподобие Internet. Создание Java и широкое распространение Internet вызвали новую вспышку интереса к интерпретаторам в целом.

Есть еще одна причина, делающая интерпретаторы интересными для исследования: они легко поддаются модификации и расширению. Если программист хочет создать свой собственный язык, с которым можно экспериментировать, то сделать это с помощью интерпретатора значительно легче, чем с помощью компилятора. Интерпретаторы лучше подходят для создания макетов оболочек программирования, потому что при их использовании правила языка можно легко изменить и быстро увидеть результат.

Интерпретатор сравнительно легко создать, понять, как он работает, легко модифицировать и, что, возможно, наиболее существенно, работать с ним увлекательно. Например, представленный в данной главе интерпретатор можно переделать таким образом, что он будет выполнять программу от конца к началу, то есть, выполнять ее, начиная с закрывающейся фигурной скобки функции `main()` и кончая открывающейся скобкой. Или можно добавить любое специальное средство языка, какое захочется программисту. Образно говоря, компиляторы предназначены для коммерческих разработок, а интерпретаторы — для свободной игры воображения. Данная глава написана именно в этом духе и автор искренне надеется, что читатель получит от нее такое же удовольствие, как и он сам при ее написании.

Определение языка Little C

Количество зарезервированных слов языка С невелико, однако это богатый и мощный язык. Чтобы описать интерпретатор полного С и его реализацию, понадобился бы значительно больший объем, чем одна глава. Интерпретатор Little C (Малый С) предназначен для интерпретации довольно узкого подмножества С, включающего, тем не менее, многие важные средства языка. При определении конкретного состава подмножества языка Little C использовались два главных критерия:

- Неотделимо ли данное средство от языка?
- Необходимо ли оно для демонстрации важных аспектов языка?

Например, такие средства, как рекурсивные функции, глобальные и локальные переменные удовлетворяют обоим критериям. Интерпретатор Little C поддерживает все три вида циклов (наличие всех их, конечно, не обязательно в соответствии с первым критерием, но необходимо в соответствии со вторым критерием). Однако оператор `switch` не включен в интерпретатор, потому что он не является обязательным (он красив, но не необходим) и не иллюстрирует ничего такого, что нельзя было бы проиллюстрировать с помощью оператора `if` (который включен в интерпретатор). Реализация оператора `switch` оставлена читателю в качестве самостоятельного упражнения.

Исходя из этих соображений, в интерпретатор Little C включены следующие средства языка:

- Параметризованные функции с локальными переменными
- Рекурсия
- Оператор `if`
- Циклы `do-while`, `while` и `for`
- Локальные и глобальные переменные типов `int` и `char`
- Параметры функций типов `int` и `char`
- Целые и символьные константы
- Строковые константы (ограниченная реализация)
- Оператор `return` (как со значением, так и без него)
- Ограниченный набор стандартных библиотечных функций
- Операторы `+`, `-`, `*`, `/`, `%`, `<`, `>`, `<=`, `>=`, `==`, `!=`, унарный `-`, унарный `+`
- Функции, возвращающие целое значение
- Комментарии вида `/*...*/`

Хоть этот набор и кажется небольшим, однако для его реализации требуется довольно объемный исходный текст программы. Одна из причин этого заключается в том, что при выполнении программы непосредственной работе интерпретатора предшествует значительная подготовительная работа программы, что обусловлено структурированностью языка.

Ограничения языка Little C

Исходный текст программы интерпретатора Little C довольно длинный, фактически, длиннее, чем следовало бы помещать в книгу. С целью упрощения этого текста в грамматику Little C введены некоторые ограничения. Первое ограничение заключается в том, что телом операторов `if`, `while`, `do` и `for` может быть только блок, заключенный в фигурные скобки. Если телом является единственный оператор, он также должен быть заключен в фигурные скобки. Например, интерпретатор Little C не сможет правильно обработать следующий фрагмент программы:

```
for(a=0; a < 10; a=a+1)
  for(b=0; b < 10; b=b+1)
    for(c=0; c < 10; c=c+1)
      puts("привет");

if(...)
  if(...) x = 10;
```

Этот фрагмент должен быть написан так:


```

for(a=0; a < 10; a=a+1) {
    for(b=0; b < 10; b=b+1) {
        for(c=0; c < 10; c=c+1) {
            puts("привет");
        }
    }
}

if(...) {
    if(...) {
        x = 10;
    }
}

```

Благодаря этому ограничению интерпретатору легче найти конец участка программы, составляющего тело одного из операторов управления программой. К тому же, поскольку чаще всего операторы управления программой обрабатывают именно блок, это ограничение не выглядит слишком обременительным. При желании читатель может самостоятельно устранить это ограничение.

Другое ограничение заключается в том, что не поддерживаются прототипы функций. Предполагается, что все функции возвращают тип `int`, разрешен возвращаемый тип `char`, но он преобразуется в `int`. Проверка правильности типа параметра не выполняется.

Все локальные переменные должны быть объявлены в самом начале функции, сразу после открывающейся фигурной скобки. Локальные переменные не могут быть объявлены внутри какого-либо блока. Поэтому следующая функция в языке Little C является неправильной:

```

int myfunc()
{
    int i; /* это допустимо */
    if(1) {
        int i; /* в языке Little C это не допустимо */
    }
}

```

Здесь объявление переменной `i` внутри блока `if` для интерпретатора Little C является недопустимым. Требование объявления локальных переменных только в начале функции немного упрощает реализацию интерпретатора. Для читателя не составит большого труда устранить это ограничение.

И, наконец, последнее ограничение: определение каждой функции должно начинаться с зарезервированного слова `char` или `int`. Следовательно, интерпретатор Little C не поддерживает традиционное правило “`int` по умолчанию”. Таким образом, следующее объявление является правильным:

```

int main()
{
    /* ... */
}

```

однако следующее объявление в языке Little C неправильное:

```

main()
{
    /* ... */
}

```

Отказ от правила “`int` по умолчанию” приближает Little C к языкам C99 и C++.

Интерпретация структурированного языка

Язык С структурирован. Это значит, что в нем определены отдельные подпрограммы с локальными переменными. В языке С также поддерживается рекурсия. Интересен тот факт, что для структурированного языка иногда легче написать компилятор, чем интерпретатор. Например, когда компилятор создает код вызова функции, он попросту заталкивает аргументы функции в системный стек и применяет к функции команду процессора CALL. При возврате функция записывает возвращаемое значение в регистр процессора, очищает стек и выполняет команду процессора RET. В то же время, если вызов функции выполняет интерпретатор, он должен на какое-то время “приостановиться”, запомнить текущее состояние, найти функцию, выполнить ее, сохранить возвращаемое значение, возвратиться в исходную точку программы и восстановить состояние, существовавшее до вызова функции. Пример выполнения этих действий будет приведен далее при рассмотрении интерпретатора. В сущности, интерпретатор должен эмулировать (выполнить другими средствами) команды процессора CALL и RET. Поддержку рекурсии также значительно легче обеспечить в компиляторе, чем в интерпретаторе.

В моей книге *The Art of C* (Berkeley, CA: Osborne/McGraw-Hill, 1991), вышедшей несколько лет назад, рассматривалась разработка интерпретатора языка small BASIC. В книге утверждается, что интерпретировать старую версию языка BASIC значительно легче, чем язык С, потому что BASIC изначально был предназначен для интерпретации. Он хорошо приспособлен к интерпретации благодаря своей неструктурированности. В нем все переменные являются глобальными и нет отдельных подпрограмм. Я по-прежнему придерживаюсь этого мнения, однако, если для интерпретатора создать средства поддержки функций, локальных переменных и рекурсии, то интерпретировать язык С станет легче, чем BASIC. Так получилось потому, что в языках типа BASIC на теоретическом уровне довольно много исключений из правил. Например, в нем знак равенства в операторе присваивания означает присваивание, а в операторе сравнения — равенство. Язык С почти полностью лишен подобных несуразностей.

На заметку

Разработанная автором реализация интерпретатора языка small BASIC содержит немало полезного для читателей, интересующихся интерпретаторами. Последняя версия интерпретатора small BASIC приведена в The C/C++ Annotated Archives (Berkeley, CA: Osborne/McGraw-Hill, 1999).

Неформальная теория языка С

Перед тем как приступить к разработке интерпретатора языка С, необходимо уяснить структуру языка С. Формальное определение языка С (например, в стандарте ANSI/ISO) очень длинное, к тому же в нем довольно много зашифрованных для неискушенного читателя положений. Однако совершенно формальное определение языка С для разработки интерпретатора не понадобится, потому что этот язык является довольно прямолинейным. Полное формальное определение языка С необходимо для создания коммерческого компилятора, а для Little С оно не является необходимым. (Фактически, в одной главе невозможно изложить формальный синтаксис, определяющий С; это заняло бы целую книгу.)

Эта глава предназначена для широкого круга читателей. Она не была задумана как формальное введение в теорию структурированных языков в целом и языка С в частности. Поэтому здесь некоторые концепции изложены упрощенно и разработка интерпретатора подмножества С ведется так, что от читателя не потребуются формальная подготовка по теории языков (структурной лингвистике).

Несмотря на это для реализации и понимания интерпретатора Little C некоторые сведения об определении языка все же необходимы. Материал, изложенный далее, является вполне достаточным для наших целей. Желая ознакомиться с более формализованным изложением материала следует обратиться к стандарту ANSI/ISO языка C.

Все программы на C представляют собой набор из одной или более функций плюс глобальные переменные (если они есть). *Функция* состоит из спецификатора типа функции, имени функции, списка параметров и блока операторов, ассоциированного с функцией. *Блок* начинается скобкой {, за которой следует последовательность из одного или нескольких операторов, и заканчивается скобкой }. Оператор языка C либо начинается с одного из зарезервированных слов, например if, либо является выражением. (Что представляет собой выражение, будет рассмотрено в следующем разделе.) Изложенные выше порождающие правила могут быть сведены в следующую таблицу:

программа	→	набор функций плюс глобальные переменные
функция	→	спецификатор список_параметров блок_операторов
блок_операторов	→	{ последовательность_операторов }
оператор	→	зарезервированное_слово, выражение или блок_операторов

Выполнение любой программы на C начинается вызовом функции main() и кончается последней закрывающейся скобкой } или первым оператором return, встретившимся в main(), если до этого не встретились exit() или abort(). Любая другая функция программы должна быть непосредственно или косвенно вызвана функцией main(). Таким образом, выполнение программы начинается с началом выполнения main() и кончается выходом из нее. Интерпретатор Little C именно так и работает.

Выражения языка C

В языке C роль выражений несколько шире, чем в других языках программирования. В общем случае в программе на C оператор может начинаться с зарезервированного слова языка C, например, while или switch, а может и не начинаться с него. Для удобства дальнейшего изложения все операторы, начинающиеся с зарезервированного слова языка C, будем называть *операторами с зарезервированным словом*. Все остальные операторы (не начинающиеся с зарезервированного слова) будем называть *операторами-выражениями*. Таким образом, все следующие операторы языка C являются операторами-выражениями:

```
count = 100; /* Строка 1 */
sample = i / 22 * (c-10); /* Строка 2 */
printf("Это выражение"); /* Строка 3 */
```

Рассмотрим каждый из этих операторов-выражений подробно. В языке C знак равенства является *оператором присваивания*¹. Здесь оператор присваивания работает не так, как, например, в BASIC. В языке BASIC значение, вычисленное в правой части знака равенства, присваивается переменной в левой части, однако, и это весьма существенно, это значение не является значением оператора присваивания. В то же время в языке C знак равенства является оператором присваивания и значение результата оператора присваивания равно значению, полученному в правой части. Следовательно, в языке C оператор присваивания фактически является *выражением присваивания*. Оператор присваивания имеет значение, поэтому он является выражением. Именно по этой причине правильными являются, например, следующие выражения:

```
a = b = c = 100;
printf("%d", a+4+5);
```

¹ Сам знак равенства является, конечно, знаком операции присваивания. — Прим. ред.

Эти выражения в языке C допустимы, потому что присваивание является оператором, имеющим значение, как и любая другая операция.

Продолжим рассмотрение предыдущего примера. Строка 2 содержит более сложное присваивание. В строке 3 вызывается функция `printf()`, выводящая на экран строку. В языке C все функции базовых типов, отличных от `void`, возвращают значение независимо от того, определен тип явно или нет. Следовательно, вызов функции, возвращающей значение, является выражением, имеющим значение, опять же независимо от того, присваивается оно чему-либо или нет. Вызов функции, не возвращающей значения (определенной со спецификатором `void`), также является выражением, однако его результат имеет тип `void`.

Определение значения выражения

Перед тем как приступить к разработке программы, способной правильно вычислить значение выражения, нужно дать более формальное определение выражения. Фактически в каждом языке программирования выражения определяются рекурсивно с помощью порождающих правил, или продукций. Интерпретатор Little C поддерживает следующие операции: `+`, `-`, `*`, `/`, `%`, `=`, операторы сравнения (`<`, `==`, `>` и так далее) и скобки. В языке Little C выражения определяются с помощью следующих порождающих правил:

выражение	→ [присваивание] [значение_переменной]
присваивание	→ именуемое_выражение = значение_переменной
именуемое_выражение	→ переменная
значение_переменной	→ часть [оператор_сравнения часть]
часть	→ терм [+терм] [-терм]
терм	→ множитель [*множитель] [/множитель] [%множитель]
множитель	→ [+ или -] атом
атом	→ переменная, константа, функция, или (выражение)

Здесь термин *оператор_сравнения* может обозначать любой из операторов сравнения. Термины *именуемое_выражение* и *значение_переменной* означают объекты в левой и правой частях оператора присваивания. Старшинство оператора определяется порождающим правилом. Чем выше старшинство оператора, тем ниже в списке операторов он расположен.

Рассмотрим применение порождающих правил на примере вычисления выражения

█ `count = 10 - 5 * 3;`

Сначала применяется правило 1, разделяющее выражение на три части:

count	=	10-5*3
↑	↑	↑
именуемое_выражение	присваивание	значение_переменной

Поскольку значение нетерминала *значение_переменной* не содержит операторов сравнения, то оно может быть сгенерировано в результате применения порождающего правила для нетерминала *терм*:

10	-	5*3
↑	↑	↑
терм	минус	терм

Несомненно, второй терм составлен из двух множителей: 5 и 3. Эти два множителя являются константами, они порождаются с помощью порождающих правил более низкого уровня.

Теперь, чтобы вычислить значение выражения, будем двигаться, следуя порождающим правилам, в обратном направлении. Сначала выполняется умножение $5 \cdot 3$, что дает 15. Потом это значение вычитается из 10, получается -5. И, наконец, последний шаг — присвоивание этого значения переменной `count`, оно же является значением всего выражения.

При создании интерпретатора Little C в первую очередь нужно построить алгоритмический эквивалент рассмотренной только что процедуры вычисления выражения.

Синтаксический анализатор выражений

Часть программы, выполняющая чтение и анализ выражения, называется *синтаксическим анализатором выражений*. Это — наиболее важная подсистема интерпретатора Little C. Так как согласно стандарту множество выражений в языке C значительно шире, чем во многих других языках программирования, то синтаксический анализатор выражений составляет значительную часть программного кода синтаксического анализатора программ.

Для построения синтаксического анализатора выражений языка C можно применить несколько различных методов. Во многих коммерческих компиляторах используются *синтаксические анализаторы, управляемые таблицей*; такие синтаксические анализаторы создаются специальными генераторами программ синтаксического анализа. Синтаксические анализаторы, управляемые таблицей, в общем случае обладают большим быстродействием, чем другие синтаксические анализаторы, однако процесс их создания очень трудоемкий. В рассматриваемом здесь интерпретаторе Little C используется *рекурсивный нисходящий синтаксический анализатор*¹, который представляет собой реализацию в языке C производящих правил, приведенных в предыдущем разделе.

Рекурсивный нисходящий синтаксический анализатор представляет собой набор взаимно рекурсивных функций, обрабатывающих выражение. Если синтаксический анализатор работает в компиляторе, то он генерирует объектный код, соответствующий исходному тексту программы. В интерпретаторе целью синтаксического анализатора является вычисление значения заданного выражения. В этом разделе рассматривается разработка синтаксического анализатора выражений языка Little C.

На заметку

Основы теории синтаксического анализа выражений рассмотрены в главе 24. Синтаксический анализатор, разрабатываемый в этой главе, строится на основе простого расширения этой теории.

Синтаксический разбор исходного текста программы

Специальная функция, читающая исходный текст программы и возвращающая очередную логическую единицу, является фундаментальной частью каждого интерпретатора и компилятора. Исторически сложившееся название такой логической единицы — *лексема*. Во всех языках программирования (в том числе и в языке C) программа рассматривается как последовательность лексем. Другими словами, лексема — это неделимая единица программы. Например, оператор равенства `==` является лексемой. Эти два знака равенства нельзя разделить, не изменив кардинальным образом их значение. Аналогично, `if` — также лексема. Ни “`i`”, ни “`f`” сами по себе не имеют в программе на C никакого значения.

В языке C каждая лексема принадлежит одной из следующих категорий:

зарезервированные слова
строки

идентификаторы
операторы

константы
знаки пунктуации

¹ Так называется программа, выполняющая синтаксический анализ методом рекурсивного спуска. — *Прим. ред.*

Зарезервированные слова — это лексемы, составляющие язык С; к ним относится, например, *while*. *Идентификаторы* — это имена переменных, функций и типов, определенных пользователем (в Little С не реализованы). *Знаки пунктуации* — это некоторые символы, такие как точка с запятой, запятая, различные скобки и т.п. В зависимости от контекста, некоторые из этих символов, могут быть операторами. Приведем пример разбиения на лексемы при разборе оператора слева направо. Оператор

```
for(x=0; x<10; x=x+1) printf("алло %d", x);
```

состоит из следующих лексем:

Лексема	Категория
for	зарезервированное слово
(знак пунктуации
x	идентификатор
=	оператор
0	константа
;	знак пунктуации
x	идентификатор
<	оператор
10	константа
;	знак пунктуации
x	идентификатор
=	оператор
x	идентификатор
+	оператор
1	константа
)	знак пунктуации
printf	идентификатор
(знак пунктуации
"алло %d"	строка
,	знак пунктуации
x	идентификатор
)	знак пунктуации
;	знак пунктуации

Однако для упрощения интерпретатора Little С в нем определяются следующие категории лексем:

Тип лексемы	Включает
delimiter (разделитель)	знаки пунктуации и операторы
keyword (зарезервированное слово)	зарезервированные слова
string (строка)	строки, заключенные в двойные кавычки
identifier (идентификатор)	имена переменных и функций
number (число)	числовая константа
block (блок)	{ или }

Функция `get_token()` выделяет лексемы из исходного текста программы Little С и возвращает их в качестве своего значения:

```
/* Считывание лексемы из входного потока. */
int get_token(void)
{
    register char *temp;

    token_type = 0; tok = 0;
```

```

temp = token;
*temp = '\0';

/* пропуск пробелов, символов табуляции и пустой строки */
while(isspace(*prog) && *prog) ++prog;

if(*prog == '\r') {
    ++prog;
    ++prog;
    /* пропуск пробела */
    while(isspace(*prog) && *prog) ++prog;
}

if(*prog == '\0') { /* конец файла */
    *token = '\0';
    tok = FINISHED;
    return (token_type = DELIMITER);
}

if(strchr("{} ", *prog)) { /* ограничители блока */
    *temp = *prog;
    temp++;
    *temp = '\0';
    prog++;
    return (token_type = BLOCK);
}

/* поиск комментариев */
if(*prog == '/')
    if(*(prog+1) == '*') { /* это комментарий */
        prog += 2;
        do { /* конец комментария */
            while(*prog != '*') prog++;
            prog++;
        } while (*prog != '/');
        prog++;
    }

if(strchr("<=>", *prog)) {
    /* возможно, это оператор сравнения */
    switch(*prog) {
        case '=': if(*(prog+1) == '=') {
            prog++; prog++;
            *temp = EQ;
            temp++; *temp = EQ; temp++;
            *temp = '\0';
        }
        break;
        case '!': if(*(prog+1) == '=') {
            prog++; prog++;
            *temp = NE;
            temp++; *temp = NE; temp++;
            *temp = '\0';
        }
        break;
        case '<': if(*(prog+1) == '=') {
            prog++; prog++;
            *temp = LE; temp++; *temp = LE;

```

```

    }
    else {
        prog++;
        *temp = LT;
    }
    temp++;
    *temp = '\0';
    break;
case '>': if(*(prog+1) == '=') {
    prog++; prog++;
    *temp = GE; temp++; *temp = GE;
}
else {
    prog++;
    *temp = GT;
}
temp++;
*temp = '\0';
break;
}
if(*token) return(token_type = DELIMITER);
}

if(strchr("+-*^/%=;(),'", *prog)){
    /* разделитель */
    *temp = *prog;
    prog++; /* продвижение на следующую позицию */
    temp++;
    *temp = '\0';
    return (token_type = DELIMITER);
}

if(*prog=="'") { /* строка в кавычках */
    prog++;
    while(*prog != "'" && *prog != '\r') *temp++ = *prog++;
    if(*prog == '\r') sintx_err(SYNTAX);
    prog++; *temp = '\0';
    return (token_type = STRING);
}

if(isdigit(*prog)) { /* число */
    while(!isdelim(*prog)) *temp++ = *prog++;
    *temp = '\0';
    return (token_type = NUMBER);
}

if(isalpha(*prog)) { /* переменная или оператор */
    while(!isdelim(*prog)) *temp++ = *prog++;
    token_type = TEMP;
}

*temp = '\0';

/* узнать, является эта строка оператором или переменной */
if(token_type==TEMP) {
    tok = look_up(token);
    /* преобразовать во внутреннее представление */
    if(tok) token_type = KEYWORD;
}

```



```

        /* это зарезервированное слово */
        else token_type = IDENTIFIER;
    }
    return token_type;
}

```

В функции `get_token()` используются следующие глобальные данные и перечислимые типы:

```

extern char *prog;
/* текущий адрес в исходном тексте программы */
extern char *p_buf;
/* указатель на начало буфера программы */

extern char token[80]; /* строковое представление лексемы */
extern char token_type; /* содержит тип лексемы */
extern char tok; /* внутреннее представление лексемы */

enum tok_types {DELIMITER, IDENTIFIER, NUMBER, KEYWORD,
                TEMP, STRING, BLOCK};

enum double_ops {LT=1, LE, GT, GE, EQ, NE};

/* Эти константы используются для вызова функции
   sntx_err() в случае синтаксической ошибки. При необходимости
   список констант можно расширить.
   ВНИМАНИЕ: константа SYNTAX используется тогда,
   когда интерпретатор не может квалифицировать ошибку.
*/
enum error_msg
{SYNTAX, UNBAL_PARENS, NO_EXP, EQUALS_EXPECTED,
 NOT_VAR, PARAM_ERR, SEMI_EXPECTED,
 UNBAL_BRACES, FUNC_UNDEF, TYPE_EXPECTED,
 NEST_FUNC, RET_NOCALL, PAREN_EXPECTED,
 WHILE_EXPECTED, QUOTE_EXPECTED, NOT_TEMP,
 TOO_MANY_LVARS, DIV_BY_ZERO};

```

Указатель `prog` указывает на текущую позицию в исходном тексте интерпретируемой программы. Указатель `p_buf` интерпретатором не изменяется; он всегда указывает на начало интерпретируемой программы. Функция `get_token()` начинает работу с удаления пробелов и символов перевода строки. Так как никакая лексема языка C (кроме строковой константы) не содержит пробелов, их нужно пропустить. Функция `get_token()` пропускает также комментарии (в Little C допускаются только комментарии вида `/*...*/`). После этого строка, представляющая каждую лексему, помещается в `token` и ее тип (определенный в перечислении `tok_types`) записывается в `token_type`. Если лексема представляет собой зарезервированное слово, то его внутреннее представление присваивается `tok` с помощью функции `look_up()` (приведена в полном листинге синтаксического анализатора). Необходимость внутреннего представления зарезервированных слов будет обоснована позже. Функция `get_token()` преобразует двухсимвольные операторы сравнения в соответствующие значения перечислимого типа. Технически в этом нет крайней необходимости, однако это упрощает реализацию интерпретатора. И, наконец, если синтаксический анализатор находит синтаксическую ошибку, то он вызывает функцию `sntx_err()` со значением перечислимого типа, соответствующим типу найденной ошибки. Функция `sntx_err()` вызывается также другими процедурами интерпретатора каждый раз, когда встречается ошибка. Листинг функции `sntx_err()` имеет такой вид:

```

/* Вывод сообщения об ошибке. */
void sntx_err(int error)
{
    char *p, *temp;
    int linecount = 0;
    register int i;

    static char *e[] = {
        "синтаксическая ошибка",
        "несбалансированные скобки",
        "выражение отсутствует",
        "ожидается знак равенства",
        "не переменная",
        "ошибка в параметре",
        "ожидается точка с запятой",
        "несбалансированные фигурные скобки",
        "функция не определена",
        "ожидается спецификатор типа",
        "слишком много вложенных вызовов функций",
        "оператор return вне функции",
        "ожидаются скобки",
        "ожидается while",
        "ожидается закрывающаяся кавычка",
        "не строка",
        "слишком много локальных переменных",
        "деление на нуль"
    };
    printf("\n%s", e[error]);
    p = p_buf;
    while(p != prog) { /* поиск номера строки с ошибкой */
        p++;
        if(*p == '\r') {
            linecount++;
        }
    }
    printf(" в строке %d\n", linecount);

    temp = p;
    for(i=0; i < 20 && p > p_buf && *p != '\n'; i++, p--);
    for(i=0; i < 30 && p <= temp; i++, p++)
        printf("%c", *p);

    longjmp(e_buf, 1); /* возврат в безопасную точку */
}

```

Обратите внимание, `sntx_err()` выводит на экран номер строки, в которой обнаружена ошибка (или номер следующей строки) и саму строку. Заканчивается `sntx_err()` вызовом `longjmp()`. Синтаксическая ошибка часто встречается внутри глубоко вложенных или рекурсивных процедур, поэтому лучшим способом реакции на ошибку является переход в какое-либо безопасное место. Как альтернативный подход, можно было бы установить глобальный флажок ошибки и просмотреть его значение во всех точках каждой процедуры, однако это существенно усложнило бы программу интерпретатора.

Рекурсивный нисходящий синтаксический анализатор Little C

Ниже приведен полный текст рекурсивного нисходящего синтаксического анализатора Little C вместе со всеми его функциями, глобальными данными и типами данных. Текст программы синтаксического анализатора находится в одном файле под

именем `PARSER.C`. (Из-за большого объема всей программы интерпретатора `Little C` она содержится в трех отдельных файлах.)

```
/* Рекурсивный нисходящий синтаксический анализатор целочисленных
   выражений, содержащих переменные и вызовы функций.
*/
#include <setjmp.h>
#include <math.h>
#include <ctype.h>
#include <stdlib.h>
#include <string.h>
#include <stdio.h>

#define NUM_FUNC      100
#define NUM_GLOBAL_VARS 100
#define NUM_LOCAL_VARS 200
#define ID_LEN        31
#define FUNC_CALLS     31
#define PROG_SIZE      10000
#define FOR_NEST        31

enum tok_types {DELIMITER, IDENTIFIER, NUMBER, KEYWORD,
                TEMP, STRING, BLOCK};

enum tokens {ARG, CHAR, INT, IF, ELSE, FOR, DO, WHILE,
             SWITCH, RETURN, EOL, FINISHED, END};

enum double_ops {LT=1, LE, GT, GE, EQ, NE};

/* Эти константы используются для вызова функции sntx_err()
   в случае синтаксической ошибки. При необходимости список констант
   можно расширить.
   ВНИМАНИЕ: константа SYNTAX используется тогда,
   когда интерпретатор не может квалифицировать ошибку.
*/
enum error_msg
{SYNTAX, UNBAL_PARENS, NO_EXP, EQUALS_EXPECTED,
 NOT_VAR, PARAM_ERR, SEMI_EXPECTED,
 UNBAL_BRACES, FUNC_UNDEF, TYPE_EXPECTED,
 NEST_FUNC, RET_NOCALL, PAREN_EXPECTED,
 WHILE_EXPECTED, QUOTE_EXPECTED, NOT_TEMP,
 TOO_MANY_LVARS, DIV_BY_ZERO};

extern char *prog; /* текущее положение в исходном тексте программы */
extern char *p_buf; /* указатель на начало буфера программы */
extern jmp_buf e_buf; /* содержит данные для longjmp() */

/* Массив этой структуры содержит информацию
   о глобальных переменных */
extern struct var_type {
    char var_name[32];
    int v_type;
    int value;
} global_vars[NUM_GLOBAL_VARS];

/* Это стек вызова функций. */
extern struct func_type {
    char func_name[32];
```

```

    int ret_type;
    char *loc; /* Адрес входа функции в файле */
} func_stack[NUM_FUNC];

/* Таблица зарезервированных слов */
extern struct commands {
    char command[20];
    char tok;
} table[];

/* Здесь функции "стандартной библиотеки"
   объявлены таким образом, что их можно
   поместить во внутреннюю таблицу функции
*/
int call_getche(void), call_putch(void);
int call_puts(void), print(void), getnum(void);

struct intern_func_type {
    char *f_name; /* имя функции */
    int (*p)(); /* указатель на функцию */
} intern_func[] = {
    "getche", call_getche,
    "putch", call_putch,
    "puts", call_puts,
    "print", print,
    "getnum", getnum,
    "", 0 /* этот список заканчивается нулем */
};

extern char token[80]; /* строковое представление лексемы */
extern char token_type; /* содержит тип лексемы */
extern char tok; /* внутреннее представление лексемы */

extern int ret_value; /* возвращаемое значение функции */

void eval_exp0(int *value);
void eval_exp(int *value);
void eval_exp1(int *value);
void eval_exp2(int *value);
void eval_exp3(int *value);
void eval_exp4(int *value);
void eval_exp5(int *value);
void atom(int *value);
void sntx_err(int error), putback(void);
void assign_var(char *var_name, int value);
int isdelim(char c), look_up(char *s), iswhite(char c);
int find_var(char *s), get_token(void);
int internal_func(char *s);
int is_var(char *s);
char *find_func(char *name);
void call(void);

/* Точка входа в синтаксический анализатор выражений. */
void eval_exp(int *value)
{
    get_token();
    if(!*token) {
        sntx_err(NO_EXP);
    }
}

```

```

    return;
}
if(*token == ';') {
    *value = 0; /* пустое выражение */
    return;
}
eval_exp0(value);
putback(); /* возврат последней лексемы во входной поток */
}

/* Обработка выражения в присваивании */
void eval_exp0(int *value)
{
    char temp[ID_LEN]; /* содержит имя переменной,
                        *   которой присваивается значение */
    register int temp_tok;

    if(token_type == IDENTIFIER) {
        if(is_var(token)) { /* если это переменная,
                             *   посмотреть, присваивается ли ей значение */
            strcpy(temp, token);
            temp_tok = token_type;
            get_token();
            if(*token == '=') { /* это присваивание */
                get_token();
                eval_exp0(value); /* вычислить присваиваемое значение */
                assign_var(temp, *value);
                /* присвоить значение */
                return;
            }
            else { /* не присваивание */
                putback(); /* восстановление лексемы */
                strcpy(token, temp);
                token_type = temp_tok;
            }
        }
    }
    eval_exp1(value);
}

/* Обработка операций сравнения. */
void eval_exp1(int *value)
{
    int partial_value;
    register char op;
    char relops[7] = {
        LT, LE, GT, GE, EQ, NE, 0
    };

    eval_exp2(value);
    op = *token;
    if(strchr(relops, op)) {
        get_token();
        eval_exp2(&partial_value);
        switch(op) { /* вычисление результата операции сравнения */
            case LT:
                *value = *value < partial_value;
                break;

```

```

        case LE:
            *value = *value <= partial_value;
            break;
        case GT:
            *value = *value > partial_value;
            break;
        case GE:
            *value = *value >= partial_value;
            break;
        case EQ:
            *value = *value == partial_value;
            break;
        case NE:
            *value = *value != partial_value;
            break;
    }
}

/* Суммирование или вычитание двух термов. */
void eval_exp2(int *value)
{
    register char op;
    int partial_value;

    eval_exp3(value);
    while((op = *token) == '+' || op == '-') {
        get_token();
        eval_exp3(&partial_value);
        switch(op) { /* суммирование или вычитание */
            case '-':
                *value = *value - partial_value;
                break;
            case '+':
                *value = *value + partial_value;
                break;
        }
    }
}

/* Умножение или деление двух множителей. */
void eval_exp3(int *value)
{
    register char op;
    int partial_value, t;

    eval_exp4(value);
    while((op = *token) == '*' || op == '/' || op == '%') {
        get_token();
        eval_exp4(&partial_value);
        switch(op) {
            /* умножение, деление или деление целых */
            case '*':
                *value = *value * partial_value;
                break;
            case '/':
                if(partial_value == 0) sntx_err(DIV_BY_ZERO);
                *value = (*value) / partial_value;
        }
    }
}

```

```

        break;
    case '%':
        t = (*value) / partial_value;
        *value = *value - (t * partial_value);
        break;
    }
}
}

/* Унарный + или - */
void eval_exp4(int *value)
{
    register char op;

    op = '\0';
    if(*token == '+' || *token == '-') {
        op = *token;
        get_token();
    }
    eval_exp5(value);
    if(op)
        if(op == '-') *value = -(*value);
}

/* Обработка выражения в скобках. */
void eval_exp5(int *value)
{
    if((*token == '(')) {
        get_token();
        eval_exp0(value); /* вычисление подвыражения */
        if(*token != ')') syntax_err(PAREN_EXPECTED);
        get_token();
    }
    else
        atom(value);
}

/* Получение значения числа, переменной или функции. */
void atom(int *value)
{
    int i;

    switch(token_type) {
    case IDENTIFIER:
        i = internal_func(token);
        if(i != -1) { /* вызов функции из стандартной библиотеки */
            *value = (*intern_func[i].p)();
        }
        else
            if(find_func(token)) {
                /* вызов функции, определенной пользователем */
                call();
                *value = ret_value;
            }
        else *value = find_var(token);
        /* получение значения переменной */
        get_token();
        return;
    }
}

```

```

case NUMBER: /* числовая константа */
    *value = atoi(token);
    get_token();
    return;
case DELIMITER:
    /* это символьная константа? */
    if(*token == '\') {
        *value = *prog;
        prog++;
        if(*prog != '\') sntx_err(QUOTE_EXPECTED);
        prog++;
        get_token();
        return ;
    }
    if(*token == '') return; /* обработка пустого выражения */
    else sntx_err(SYNTAX); /* синтаксическая ошибка */
default:
    sntx_err(SYNTAX); /* синтаксическая ошибка */
}
}

/* Вывод сообщения об ошибке. */
void sntx_err(int error)
{
    char *p, *temp;
    int linecount = 0;
    register int i;

    static char *e[] = {
        "синтаксическая ошибка",
        "несбалансированные скобки",
        "выражение отсутствует",
        "ожидается знак равенства",
        "не переменная",
        "ошибка в параметре",
        "ожидается точка с запятой",
        "несбалансированные фигурные скобки",
        "функция не определена",
        "ожидается спецификатор типа",
        "слишком много вложенных вызовов функций",
        "возврат без вызова",
        "ожидаются скобки",
        "ожидается while",
        "ожидается закрывающаяся кавычка",
        "не строка",
        "слишком много локальных переменных",
        "деление на нуль"
    };

    printf("\n%s", e[error]);
    p = p_buf;
    while(p != prog) { /* поиск номера строки с ошибкой */
        p++;
        if(*p == '\r') {
            linecount++;
        }
    }
    printf(" в строке %d\n", linecount);
}

```



```

temp = p;
for(i=0; i < 20 && p > p_buf && *p != '\n'; i++, p--);
for(i=0; i < 30 && p <= temp; i++, p++)
    printf("%c", *p);

longjmp(e_buf, 1); /* возврат в безопасную точку */
}

/* Считывание лексемы из входного потока. */
int get_token(void)
{
    register char *temp;

    token_type = 0; tok = 0;

    temp = token;
    *temp = '\0';

    /* пропуск пробелов, символов табуляции и пустой строки */
    while(isspace(*prog) && *prog) ++prog;

    if(*prog == '\r') {
        ++prog;
        ++prog;
        /* пропуск пробела */
        while(isspace(*prog) && *prog) ++prog;
    }

    if(*prog == '\0') { /* конец файла */
        *token = '\0';
        tok = FINISHED;
        return (token_type = DELIMITER);
    }

    if(strchr("{} ", *prog)) { /* ограничители блока */
        *temp = *prog;
        temp++;
        *temp = '\0';
        prog++;
        return (token_type = BLOCK);
    }

    /* поиск комментариев */
    if(*prog == '/')
        if(*(prog+1) == '*') { /* это комментарий */
            prog += 2;
            do { /* найти конец комментария */
                while(*prog != '*') prog++;
                prog++;
            } while (*prog != '/');
            prog++;
        }

    if(strchr("<=> ", *prog)) {
        /* возможно, это оператор сравнения */
        switch(*prog) {
            case '=': if(*(prog+1) == '=') {

```

```

        prog++; prog++;
        *temp = EQ;
        temp++; *temp = EQ; temp++;
        *temp = '\0';
    }
    break;
    case '!': if(*(prog+1) == '=') {
        prog++; prog++;
        *temp = NE;
        temp++; *temp = NE; temp++;
        *temp = '\0';
    }
    break;
    case '<': if(*(prog+1) == '=') {
        prog++; prog++;
        *temp = LE; temp++; *temp = LE;
    }
    else {
        prog++;
        *temp = LT;
    }
    temp++;
    *temp = '\0';
    break;
    case '>': if(*(prog+1) == '=') {
        prog++; prog++;
        *temp = GE; temp++; *temp = GE;
    }
    else {
        prog++;
        *temp = GT;
    }
    temp++;
    *temp = '\0';
    break;
}
if(*token) return(token_type = DELIMITER);
}

if(strchr("+-*^/%=;(),'", *prog)){
    /* разделитель */
    *temp = *prog;
    prog++; /* продвижение на следующую позицию */
    temp++;
    *temp = '\0';
    return (token_type = DELIMITER);
}

if(*prog=="'") { /* строка в кавычках */
    prog++;
    while(*prog != '"' && *prog != '\r') *temp++ = *prog++;
    if(*prog == '\r') sntx_err(SYNTAX);
    prog++; *temp = '\0';
    return (token_type = STRING);
}

if(isdigit(*prog)) { /* число */
    while(!isdelim(*prog)) *temp++ = *prog++;

```

```

        *temp = '\0';
        return (token_type = NUMBER);
    }

    if(isalpha(*prog)) { /* переменная или оператор */
        while(!isdelim(*prog)) *temp++ = *prog++;
        token_type = TEMP;
    }

    *temp = '\0';

    /* эта строка является оператором или переменной? */
    if(token_type==TEMP) {
        tok = look_up(token);
        /* преобразовать во внутреннее представление */
        if(tok) token_type = KEYWORD;
        /* это зарезервированное слово */
        else token_type = IDENTIFIER;
    }
    return token_type;
}

/* Возврат лексемы во входной поток. */
void putback(void)
{
    char *t;

    t = token;
    for(; *t; t++) prog--;
}

/* Поиск внутреннего представления лексемы
   в таблице лексем.
*/
int look_up(char *s)
{
    register int i;
    char *p;

    /* преобразование в нижний регистр */
    p = s;
    while(*p) { *p = tolower(*p); p++; }

    /* Есть ли лексема в таблице? */
    for(i=0; *table[i].command; i++) {
        if(!strcmp(table[i].command, s)) return table[i].tok;
    }
    return 0; /* незнакомый оператор */
}

/* Возвращает индекс функции во внутренней
   библиотеке, или -1, если не найдена.
*/
int internal_func(char *s)
{
    int i;

    for(i=0; intern_func[i].f_name[0]; i++) {

```

```

    if(!strcmp(intern_func[i].f_name, s)) return i;
}
return -1;
}

/* Возвращает true (ИСТИНА), если с - разделитель. */
int isdelim(char c)
{
    if(strchr(" !,+-<>'/*%^=()", c) || c == 9 ||
       c == '\r' || c == 0) return 1;
    return 0;
}

/* Возвращает 1, если с - пробел
или табуляция. */
int iswhite(char c)
{
    if(c == ' ' || c == '\t') return 1;
    else return 0;
}

```

Функции, начинающиеся с `eval_exp`, и функция `atom()` реализуют порождающие правила для выражений в Little C. Для их проверки (и в качестве упражнения) рекомендуется мысленно выполнить действия синтаксического анализатора для какого-либо простого выражения.

Функция `atom()` находит значение целой константы или переменной, функции или символьной константы. В тексте программы могут присутствовать функции двух видов: определенные пользователем и библиотечные. Если встретилась пользовательская функция, ее текст обрабатывается интерпретатором до получения возвращаемого значения и выхода из функции. (Вызов функции рассматривается в следующем разделе.) Если встретилась библиотечная функция, то сначала ищется ее адрес с помощью функции `internal_func()`, а затем устанавливается доступ к ней с помощью ее интерфейсной функции. Библиотечные функции и адреса их интерфейсных функций содержатся в массиве `intern_func`, приведенном ниже:

```

/* Здесь функции "стандартной библиотеки" объявлены таким образом,
что их можно поместить во внутреннюю таблицу функции.
*/
int call_getche(void), call_putch(void);
int call_puts(void), print(void), getnum(void);

struct intern_func_type {
    char *f_name; /* имя функции */
    int (*p)(); /* указатель на функцию */
} intern_func[] = {
    "getche", call_getche,
    "putch", call_putch,
    "puts", call_puts,
    "print", print,
    "getnum", getnum,
    "", 0 /* этот список заканчивается нулем */
};

```

Таким образом, в интерпретаторе Little C предусмотрено только несколько функций стандартной библиотеки, однако расширить их список очень легко. (Тексты интерфейсных функций содержатся в отдельном файле, рассматриваемом далее в разделе "Библиотечные функции Little C".)

Необходимо сделать еще одно замечание о процедурах в файле синтаксического анализатора. Для правильного анализа программы на С в некоторых случаях требуется так называемый *просмотр на одну лексему вперед*. Например, в операторе

```
alpha = count();
```

интерпретатор сможет определить, что `count` является функцией, а не переменной, только если просмотрит на одну лексему вперед, то есть прочтет следующую скобку. Однако, если оператор выглядит как

```
alpha = count * 10;
```

то следующую после `count` лексему (в данном случае `*`) нужно вернуть обратно во входной поток; она будет использована позднее. Поэтому в файл синтаксического анализатора выражений включена функция `putback()`, которая возвращает последнюю прочитанную лексему обратно во входной поток.

В файле синтаксического анализатора выражений могут встретиться функции, в данный момент непонятные для читателя, однако в процессе изучения Little C их назначение и принцип работы станут яснее.

Интерпретатор Little C

В этом разделе рассматривается наиболее важная часть интерпретатора Little C. Перед тем как приступить к подробному чтению текста программы интерпретатора, нужно понять, как вообще работает интерпретатор. Понять программу интерпретатора в некотором смысле легче, чем программу синтаксического анализатора выражений, потому что работа по интерпретации программы на С может быть выражена следующим простым алгоритмом:

```
while (есть_лексема_во_входном_потоке) {  
    читать следующую лексему;  
    выполнить соответствующее действие;  
}
```

Этот алгоритм может показаться невероятно простым по сравнению с синтаксическим анализатором выражений, но это именно то, что делает интерпретатор. Нужно только иметь в виду следующее: шаг “выполнить соответствующее действие” может содержать чтение дополнительных лексем из входного потока. Для лучшего понимания этого алгоритма мысленно выполним интерпретацию следующего фрагмента программы:

```
int a;  
  
a = 10;  
  
if(a < 100) printf("%d", a);
```

Согласно алгоритму, прочтем первую лексему `int`. Эта лексема указывает на то, что следующим действием должно быть чтение следующей лексемы для того, чтобы узнать, как называется переменная (`a`), которую нужно объявить и для которой нужно выделить область памяти. Следующая лексема (точка с запятой) заканчивает строку. Соответствующее действие — проигнорировать ее. Далее, начинаем следующую итерацию алгоритма и считываем следующую лексему, это `a` из второй строки. Строка не начинается с зарезервированного слова, следовательно, это выражение языка С. Поэтому соответствующим действием является применение синтаксического анализатора выражений для вычисления значения выражения. Этот процесс “съедает” все лексемы во второй строке. Наконец, читаем лексему `if`. Она указывает на то, что начинается оператор `if`. Соответствующее действие — выполнить его. Аналогичный процесс вы-

полняется многократно, пока не будет считана последняя лексема программы. Это относится к любой программе на С. Пользуясь этим алгоритмом, приступим к созданию интерпретатора.

Предварительный проход интерпретатора

Перед тем как интерпретатор начнет выполнять программу, должны быть выполнены некоторые рутинные процедуры. Характерной чертой языков, предназначенных больше для интерпретации, чем для компиляции, является то, что выполнение программы начинается в начале текста программы и заканчивается в его конце. Так выполняются программы, написанные на старых версиях языка BASIC. Это, однако, не относится к языку С (как и к любому другому структурированному языку) по трем основным причинам.

Во-первых, все программы на С начинают выполняться с функции `main()`. Совсем не обязательно, чтобы эта функция была первой в программе. Поэтому интерпретатор, чтобы начать выполнение с нее, должен еще до начала выполнения программы узнать, где она находится. Следовательно, должен быть реализован некоторый метод, позволяющий начать выполнение программы с нужной точки. (Глобальные переменные также могут предшествовать функции `main()`, поэтому, даже если она является первой функцией программы, все равно и в этом случае она не начинается с первой строки.)

Во-вторых, все глобальные переменные должны быть известны перед началом выполнения `main()`. Операторы объявления глобальных переменных никогда не выполняются интерпретатором, потому что они находятся вне всех функций. (В языке С весь выполняющийся текст программы находится *внутри* функций, поэтому при выполнении программы интерпретатор Little С никогда не выходит за пределы функций.)

И наконец, в-третьих, для повышения скорости выполнения необходимо (правда, не всегда) знать, где в программе расположена каждая функция; это позволит вызывать ее как можно быстрее. Если это условие не будет выполнено, то при каждом вызове функции понадобится длительный последовательный поиск этой функции в тексте программы.

Эти проблемы решаются с помощью *предварительного прохода интерпретатора*. Программа предварительного прохода (иногда ее называют препроцессором, правда это название очень неудачное из-за того, что совпадает с названием препроцессора компилятора С, хотя практически ничего общего с ним не имеет) применяется во всех коммерческих компиляторах независимо от интерпретируемого языка. Программа предварительного прохода читает исходный текст программы перед ее выполнением и делает все, что нужно сделать до выполнения. В интерпретаторе Little С она выполняет две важные задачи: во-первых, находит и запоминает положение всех пользовательских функций, включая `main()`, и во-вторых, находит все глобальные переменные и определяет область их видимости. В интерпретаторе Little С предварительный проход выполняет функция `prescan()`:

```
/* Поиск всех функций программы
   и размещение глобальных переменных. */
void prescan(void)
{
    char *p, *tp;
    char temp[32];
    int datatype;
    int brace = 0; /* Если brace равно 0, то
                     текущая позиция указателя программы вне
                     какой-либо функции. */

    p = prog;
```

```

func_index = 0;
do {
    while(brace) { /* обход кода функции */
        get_token();
        if(*token == '{') brace++;
        if(*token == '}') brace--;
    }

    tp = prog; /* запоминание текущей позиции */
    get_token();
    /* тип глобальной переменной или возвращаемого значения функции */
    if(tok==CHAR || tok==INT) {
        datatype = tok; /* запоминание типа данных */
        get_token();
        if(token_type == IDENTIFIER) {
            strcpy(temp, token);
            get_token();
            if(*token != '(') { /* должна быть глобальная переменная */
                prog = tp;
                /* возврат в начало объявления */
                decl_global();
            }
            else if(*token == '(') { /* должна быть функция */
                func_table[func_index].loc = prog;
                func_table[func_index].ret_type = datatype;
                strcpy(func_table[func_index].func_name, temp);
                func_index++;
                while(*prog != ')') prog++;
                prog++;
                /* сейчас prog указывает на открывающуюся
                 * фигурную скобку функции */
            }
            else putback();
        }
    }
    else if(*token == '{') brace++;
} while(tok != FINISHED);
prog = p;
}

```

Функция `prescan()` работает следующим образом. Каждый раз, когда встречается открывающаяся фигурная скобка, переменная `brace` увеличивается на 1, а когда закрывающаяся — уменьшается на 1. Следовательно, если `brace` больше нуля, то текущая лексема находится внутри функции¹. Поэтому объявление переменной считается глобальным, если оно встретилось, когда `brace` равно нулю. Аналогично, если при `brace`, равном нулю, встретилось имя функции, значит, оно принадлежит определению функции (в Little C нет прототипов функций).

Функция `decl_global()` запоминает глобальные переменные в таблице `global_vars`:

```

/* Массив этих структур содержит информацию
   о глобальных переменных.
*/

```

¹ На самом деле, конечно, данный алгоритм работает правильно только при условии, что учитываются только значащие фигурные скобки, а фигурные скобки внутри строк, например, не учитываются. (Фигурные скобки внутри строк “съедаются” программой считывания лексем.) — *Прим. ред.*

```

struct var_type {
    char var_name[ID_LEN];
    int v_type;
    int value;
} global_vars[NUM_GLOBAL_VARS];

int gvar_index; /* индекс в таблице глобальных переменных */

/* Объявление глобальной переменной. */
void decl_global(void)
{
    int vartype;

    get_token(); /* определение типа */

    vartype = tok; /* запоминание типа переменной */

    do { /* обработка списка */
        global_vars[gvar_index].v_type = vartype;
        global_vars[gvar_index].value = 0; /* инициализация нулем */
        get_token(); /* определение имени */
        strcpy(global_vars[gvar_index].var_name, token);
        get_token();
        gvar_index++;
    } while(*token == ',');
    if(*token != ';') sintx_err(SEMI_EXPECTED);
}

```

Переменная целого типа `gvar_index` содержит индекс первого свободного элемента массива `global_vars`.

Адрес каждой функции, определенной пользователем, помещается в массив `func_table`:

```

struct func_type {
    char func_name[ID_LEN];
    int ret_type;
    char *loc; /* адрес точки входа в файле */
} func_table[NUM_FUNC];
int func_index; /* индекс в таблице функций */

```

Переменная `func_index` содержит индекс первой свободной позиции в таблице `func_table`.

Функция `main()`

Главная функция интерпретатора Little C загружает исходный текст программы, инициализирует глобальные переменные, готовит интерпретатор к вызову `main()` и вызывает функцию `call()`, которая начинает выполнение программы. Работа `call()` будет рассмотрена далее в этой главе.

```

int main(int argc, char *argv[])
{
    if(argc != 2) {
        printf("Применение: littlec <имя_файла>\n");
        exit(1);
    }

    /* выделение памяти для программы */
    if((p_buf = (char *) malloc(PROG_SIZE)) == NULL) {

```



```

    printf("Выделить память не удалось.");
    exit(1);
}

/* загрузка программы для выполнения */
if(!load_program(p_buf, argv[1])) exit(1);
if(setjmp(e_buf)) exit(1);
/* инициализация буфера long jump */

gvar_index = 0;
/* инициализация индекса глобальных переменных */

/* установка указателя программы
   в начало буфера программы */
prog = p_buf;
prescan(); /* найти все функции
            и глобальные переменные в программе */

lvartos = 0; /* инициализация индекса стека локальных переменных */
functos = 0; /* инициализация индекса стека вызовов (CALL) */

/* первой вызывается функция main() */
prog = find_func("main"); /* найти точку входа программы */

if(!prog) { /* функция main() неправильна или отсутствует */
    printf("main() не найдена.\n");
    exit(1);
}

prog--; /* возврат к открывающей скобке ( */
strcpy(token, "main");
call(); /* начало интерпретации main() */

return 0;
}

```

Функция interp_block()

Функция `interp_block()` является сердцем интерпретатора. В этой функции принимается решение о том, какое действие выполнить при прочтении очередной лексемы из входного потока. Функция интерпретирует один блок программы, после чего возвращает управление. Если блок состоит из единственного оператора, этот оператор интерпретируется и функция возвращает управление вызвавшей программе. По умолчанию `interp_block()` интерпретирует один оператор, после чего возвращает управление вызвавшей программе. Однако, если встречается открывающаяся фигурная скобка, то флажок `block` устанавливается в 1 и функция продолжает интерпретацию операторов, пока не встретит закрывающуюся фигурную скобку. Текст функции `interp_block()` приведен ниже:

```

/* Интерпретация одного оператора или блока. Когда
   interp_block возвращает управление после первого вызова,
   в main() была найдена последняя закрывающаяся
   фигурная скобка или оператор return.
*/
void interp_block(void)
{
    int value;

```

```

char block = 0;

do {
    token_type = get_token();

    /* При интерпретации одного оператора управление
       возвращается, как только встретилась первая точка с запятой.
    */

    /* определение типа лексемы */
    if(token_type == IDENTIFIER) {
        /* Это не зарезервированное слово,
           обрабатывается выражение. */
        putback(); /* возврат лексемы во входной поток
                     для дальнейшей обработки функцией eval_exp() */
        eval_exp(&value); /* обработка выражения */
        if(*token!=';') syntax_err(SEMI_EXPECTED);
    }
    else if(token_type==BLOCK) { /* если ограничитель блока */
        if(*token == '{') /* блок */
            block = 1;
        /* интерпретация блока, а не оператора */
        else return; /* это }, возврат из функции */
    }
    else /* это зарезервированное слово */
        switch(tok) {
            case CHAR:
            case INT:
                /* объявление локальной переменной */
                putback();
                decl_local();
                break;
            case RETURN: /* возврат из функции */
                func_ret();
                return;
            case IF: /* обработка оператора if */
                exec_if();
                break;
            case ELSE: /* обработка оператора else */
                find_eob(); /* поиск конца блока else
                             и продолжение выполнения */
                break;
            case WHILE: /* обработка цикла while */
                exec_while();
                break;
            case DO: /* обработка цикла do-while */
                exec_do();
                break;
            case FOR: /* обработка цикла for */
                exec_for();
                break;
            case END:
                exit(0);
        }
    } while (tok != FINISHED && block);
}

```

Если не считать вызовов функции `exit()` или подобных ей, то интерпретация программы, написанной на языке С, кончается в одном из следующих случаев: встретилась последняя закрывающаяся фигурная скобка функции `main()`, или встретился оператор `return` из `main()`. Из-за того, что при встрече последней закрывающейся фигурной скобки `main()` программу нужно завершить, `interp_block()` выполняет только один оператор или блок, а не всю программу, хоть она и состоит из блоков. Таким образом, `interp_block()` вызывается каждый раз, когда встречается новый блок. Это относится не только к блокам функций, но и к блокам операторов (например, `if`). Следовательно, в процессе выполнения программы интерпретатор Little C вызывает `interp_block()` рекурсивно.

Функция `interp_block()` работает следующим образом. Сначала из входного потока считывается очередная лексема программы. Если это точка с запятой, то выполняется единственный оператор и функция возвращает управление. В противном случае, выполняется проверка, является ли следующая лексема идентификатором; если да, то оператор является выражением и вызывается синтаксический анализатор выражений. Синтаксический анализатор должен прочесть все выражение, включая первую лексему, поэтому перед его вызовом функция `putback()` возвращает последнюю прочитанную лексему во входной поток. После возврата управления из `eval_exp()` `token` содержит последнюю лексему, прочитанную синтаксическим анализатором выражений. Если синтаксических ошибок нет, то это должна быть точка с запятой. Если `token` не содержит точку с запятой, то выводится сообщение об ошибке.

Если очередная лексема программы является открывающейся фигурной скобкой, то переменная `block` устанавливается равной 1, а если закрывающейся, то `interp_block()` возвращает управление вызвавшей программе.

Если лексема является зарезервированным словом, то выполняется оператор `switch`, который вызывает соответствующую процедуру обработки оператора. Нумерация зарезервированных слов в функции `get_token()` нужна для того, чтобы можно было применить `switch`, а не `if`, которому пришлось бы сравнивать строки, что значительно медленнее.

Функции, выполняющие операторы с зарезервированным словом, будут рассмотрены в дальнейших разделах.

Ниже приведен листинг файла с текстом программы интерпретатора. Он называется `LITTLE.C`.

```
/* Интерпретатор языка Little C. */

#include <stdio.h>
#include <setjmp.h>
#include <math.h>
#include <ctype.h>
#include <stdlib.h>
#include <string.h>

#define NUM_FUNC          100
#define NUM_GLOBAL_VARS  100
#define NUM_LOCAL_VARS   200
#define NUM_BLOCK        100
#define ID_LEN           31
#define FUNC_CALLS       31
#define NUM_PARAMS       31
#define PROG_SIZE        10000
#define LOOP_NEST        31

enum tok_types {DELIMITER, IDENTIFIER, NUMBER, KEYWORD,
                TEMP, STRING, BLOCK};
```

```

/* сюда можно добавить дополнительные лексемы зарезервированных слов */
enum tokens {ARG, CHAR, INT, IF, ELSE, FOR, DO, WHILE,
             SWITCH, RETURN, EOL, FINISHED, END};

/* сюда можно добавить дополнительные двухсимвольные операторы,
   например, -> */
enum double_ops {LT=1, LE, GT, GE, EQ, NE};

/* Это константы вызова sntx_err() при
   синтаксической ошибке. Их список можно расширить.
   Обратите внимание: SYNTAX представляет собой
   нераспознанную ошибку.
*/
enum error_msg
{SYNTAX, UNBAL_PARENS, NO_EXP, EQUALS_EXPECTED,
 NOT_VAR, PARAM_ERR, SEMI_EXPECTED,
 UNBAL_BRACES, FUNC_UNDEF, TYPE_EXPECTED,
 NEST_FUNC, RET_NOCALL, PAREN_EXPECTED,
 WHILE_EXPECTED, QUOTE_EXPECTED, NOT_TEMP,
 TOO_MANY_LVARS, DIV_BY_ZERO};

char *prog; /* текущая позиция в исходном тексте программы */
char *p_buf; /* указывает на начало буфера программы */
jmp_buf e_buf; /* содержит информацию для longjmp() */

/* Массив этих структур содержит информацию
   о глобальных переменных.
*/
struct var_type {
    char var_name[ID_LEN];
    int v_type;
    int value;
} global_vars[NUM_GLOBAL_VARS];

struct var_type local_var_stack[NUM_LOCAL_VARS];

struct func_type {
    char func_name[ID_LEN];
    int ret_type;
    char *loc; /* адрес точки входа в файле */
} func_table[NUM_FUNC];

int call_stack[NUM_FUNC];

struct commands { /* таблица зарезервированных слов */
    char command[20];
    char tok;
} table[] = { /* В эту таблицу */
    "if", IF, /* команды должны быть введены на нижнем регистре. */
    "else", ELSE,
    "for", FOR,
    "do", DO,
    "while", WHILE,
    "char", CHAR,
    "int", INT,
    "return", RETURN,
    "end", END,
    "", END /* конец таблицы */
}

```

```

};

char token[80];
char token_type, tok;

int functos; /* индекс вершины стека вызова функций */
int func_index; /* индекс в таблице функций */
int gvar_index; /* индекс в таблице глобальных переменных */
int lvar_tos; /* индекс в стеке локальных переменных */

int ret_value; /* возвращаемое значение функции */

void print(void), prescan(void);
void decl_global(void), call(void), putback(void);
void decl_local(void), local_push(struct var_type i);
void eval_exp(int *value), syntax_err(int error);
void exec_if(void), find_eob(void), exec_for(void);
void get_params(void), get_args(void);
void exec_while(void), func_push(int i), exec_do(void);
void assign_var(char *var_name, int value);
int load_program(char *p, char *fname),
    find_var(char *s);
void interp_block(void), func_ret(void);
int func_pop(void), is_var(char *s), get_token(void);
char *find_func(char *name);

int main(int argc, char *argv[])
{
    if(argc != 2) {
        printf("Применение: littlec <имя_файла>\n");
        exit(1);
    }

    /* выделение памяти для программы */
    if((p_buf = (char *) malloc(PROG_SIZE))==NULL) {
        printf("Выделить память не удалось.");
        exit(1);
    }

    /* загрузка программы для выполнения */
    if(!load_program(p_buf, argv[1])) exit(1);
    if(setjmp(e_buf)) exit(1);
    /* инициализация буфера long jump */

    gvar_index = 0; /* инициализация индекса глобальных переменных */

    /* установка указателя программы на начало буфера программы */
    prog = p_buf;
    prescan(); /* определение адресов всех функций
               и глобальных переменных программы */

    lvar_tos = 0; /* инициализация индекса стека локальных переменных */
    functos = 0; /* инициализация индекса стека вызовов (CALL) */

    /* первой вызывается main() */
    prog = find_func("main"); /* поиск точки входа программы */

    if(!prog) { /* функция main() неправильна или отсутствует */

```

```

    printf("main() не найдена.\n");
    exit(1);
}

prog--; /* возврат к открывающейся скобке ( */
strcpy(token, "main");
call(); /* начало интерпретации main() */

return 0;
}

/* Интерпретация одного оператора или блока. Когда
interp_block возвращает управление после первого
вызова, в main() встретилась последняя
закрывающаяся фигурная скобка или оператор return.
*/
void interp_block(void)
{
    int value;
    char block = 0;

    do {
        token_type = get_token();

        /* При интерпретации одного оператора возврат
        управления после первой точки с запятой.
        */

        /* определение типа лексемы */
        if(token_type == IDENTIFIER) {
            /* Это не зарезервированное слово,
            обрабатывается выражение. */
            putback(); /* возврат лексемы во входной поток
            для дальнейшей обработки функцией eval_exp() */
            eval_exp(&value); /* обработка выражения */
            if(*token!=';') syntax_err(SEMI_EXPECTED);
        }
        else if(token_type==BLOCK) {
            /* если это ограничитель блока */
            if(*token == '{') /* блок */
                block = 1;
            /* интерпретация блока, а не оператора */
            else return; /* это }, возврат */
        }
        else /* зарезервированное слово */
            switch(tok) {
                case CHAR:
                case INT: /* объявление локальной переменной */
                    putback();
                    decl_local();
                    break;
                case RETURN: /* возврат из вызова функции */
                    func_ret();
                    return;
                case IF: /* обработка оператора if */
                    exec_if();
                    break;
                case ELSE: /* обработка оператора else */

```

```

        find_eob(); /* поиск конца блока else
                   и продолжение выполнения */
        break;
    case WHILE: /* обработка цикла while */
        exec_while();
        break;
    case DO: /* обработка цикла do-while */
        exec_do();
        break;
    case FOR: /* обработка цикла for */
        exec_for();
        break;
    case END:
        exit(0);
    }
} while (tok != FINISHED && block);
}

/* Загрузка программы. */
int load_program(char *p, char *fname)
{
    FILE *fp;
    int i=0;

    if((fp=fopen(fname, "rb"))==NULL) return 0;

    i = 0;
    do {
        *p = getc(fp);
        p++; i++;
    } while(!feof(fp) && i<PROG_SIZE);

    if(*(p-2) == 0x1a) *(p-2) = '\0'; /* программа кончается нулевым
                                     символом */
    else *(p-1) = '\0';
    fclose(fp);
    return 1;
}

/* Найти адреса всех функций в программе
   и запомнить глобальные переменные. */
void prescan(void)
{
    char *p, *tp;
    char temp[32];
    int datatype;
    int brace = 0; /* Если brace = 0, то текущая
                   позиция указателя программы находится
                   вне какой-либо функции. */

    p = prog;
    func_index = 0;
    do {
        while(brace) { /* обход кода функции */
            get_token();
            if(*token == '{') brace++;
            if(*token == '}') brace--;
        }
    }
}

```

```

tp = prog; /* запоминание текущей позиции */
get_token();
/* тип глобальной переменной или возвращаемого значения функции */
if(tok==CHAR || tok==INT) {
    datatype = tok; /* запоминание типа данных */
    get_token();
    if(token_type == IDENTIFIER) {
        strcpy(temp, token);
        get_token();
        if(*token != '(') { /* это должна быть глобальная переменная */
            prog = tp;
            /* возврат в начало объявления */
            decl_global();
        }
        else if(*token == '(') { /* это должна быть функция */
            func_table[func_index].loc = prog;
            func_table[func_index].ret_type = datatype;
            strcpy(func_table[func_index].func_name, temp);
            func_index++;
            while(*prog != ')') prog++;
            prog++;
            /* сейчас prog указывает на открывающуюся
            фигурную скобку функции */
        }
        else putback();
    }
}
else if(*token == '{') brace++;
} while(tok != FINISHED);
prog = p;
}

/* Возврат адреса точки входа данной функции.
Возврат NULL, если не найдена.
*/
char *find_func(char *name)
{
    register int i;

    for(i=0; i < func_index; i++)
        if(!strcmp(name, func_table[i].func_name))
            return func_table[i].loc;

    return NULL;
}

/* Объявление глобальной переменной. */
void decl_global(void)
{
    int vartype;

    get_token(); /* определение типа */

    vartype = tok; /* запоминание типа переменной */

    do { /* обработка списка */
        global_vars[gvar_index].v_type = vartype;

```



```

    global_vars[gvar_index].value = 0; /* инициализация нулем */
    get_token(); /* определение имени */
    strcpy(global_vars[gvar_index].var_name, token);
    get_token();
    gvar_index++;
} while(*token == ',');
if(*token != ';') syntax_err(SEMI_EXPECTED);
}

/* Объявление локальной переменной. */
void decl_local(void)
{
    struct var_type i;

    get_token(); /* определение типа */

    i.v_type = tok;
    i.value = 0; /* инициализация нулем */

    do { /* обработка списка */
        get_token(); /* определение имени переменной */
        strcpy(i.var_name, token);
        local_push(i);
        get_token();
    } while(*token == ',');
    if(*token != ';') syntax_err(SEMI_EXPECTED);
}

/* Вызов функции. */
void call(void)
{
    char *loc, *temp;
    int lvartemp;

    loc = find_func(token); /* найти точку входа функции */
    if(loc == NULL)
        syntax_err(FUNC_UNDEF); /* функция не определена */
    else {
        lvartemp = lvartos; /* запоминание индекса стека локальных
                               переменных */
        get_args(); /* получение аргументов функции */
        temp = prog; /* запоминание адреса возврата */
        func_push(lvartemp); /* запоминание индекса стека локальных
                               переменных */
        prog = loc; /* переустановка prog в начало функции */
        get_params(); /* загрузка параметров функции
                        значениями аргументов */
        interp_block(); /* интерпретация функции */
        prog = temp; /* восстановление prog */
        lvartos = func_pop();
        /* восстановление стека локальных переменных */
    }
}

/* Заталкивание аргументов функции в стек
   локальных переменных. */
void get_args(void)
{

```

```

int value, count, temp[NUM_PARAMS];
struct var_type i;

count = 0;
get_token();
if(*token != '(') sntx_err(PAREN_EXPECTED);

/* обработка списка значений */
do {
    eval_exp(&value);
    temp[count] = value; /* временное запоминание */
    get_token();
    count++;
}while(*token == ',');
count--;
/* затолкнуть в local_var_stack в обратном порядке */
for(; count>=0; count--) {
    i.value = temp[count];
    i.v_type = ARG;
    local_push(i);
}
}

/* Получение параметров функции. */
void get_params(void)
{
    struct var_type *p;
    int i;

    i = lvar_tos-1;
    do { /* обработка списка параметров */
        get_token();
        p = &local_var_stack[i];
        if(*token != ')') {
            if(tok != INT && tok != CHAR)
                sntx_err(TYPE_EXPECTED);

            p->v_type = token_type;
            get_token();

            /* связывание имени параметра с аргументом,
               уже находящимся в стеке локальных переменных */
            strcpy(p->var_name, token);
            get_token();
            i--;
        }
        else break;
    } while(*token == ',');
    if(*token != ')') sntx_err(PAREN_EXPECTED);
}

/* Возврат из функции. */
void func_ret(void)
{
    int value;

    value = 0;
    /* получение возвращаемого значения, если оно есть */

```

```

    eval_exp(&value);

    ret_value = value;
}

/* Затолкнуть локальную переменную. */
void local_push(struct var_type i)
{
    if(lvartos > NUM_LOCAL_VARS)
        sntx_err(TOO_MANY_LVARS);

    local_var_stack[lvartos] = i;
    lvartos++;
}

/* Выталкивание индекса в стеке локальных переменных. */
int func_pop(void)
{
    functos--;
    if(functos < 0) sntx_err(RET_NOCALL);
    return call_stack[functos];
}

/* Запись индекса в стек локальных переменных. */
void func_push(int i)
{
    if(functos > NUM_FUNC)
        sntx_err(NEST_FUNC);
    call_stack[functos] = i;
    functos++;
}

/* Присваивание переменной значения. */
void assign_var(char *var_name, int value)
{
    register int i;

    /* проверка наличия локальной переменной */
    for(i=lvartos-1; i >= call_stack[functos-1]; i--) {
        if(!strcmp(local_var_stack[i].var_name, var_name)) {
            local_var_stack[i].value = value;
            return;
        }
    }
    if(i < call_stack[functos-1])
        /* если переменная не локальная, ищем ее в таблице глобальных
        переменных */
        for(i=0; i < NUM_GLOBAL_VARS; i++)
            if(!strcmp(global_vars[i].var_name, var_name)) {
                global_vars[i].value = value;
                return;
            }
    sntx_err(NOT_VAR); /* переменная не найдена */
}

/* Получение значения переменной. */
int find_var(char *s)
{

```

```

register int i;

/* проверка наличия переменной */
for(i=lvartos-1; i >= call_stack[functos-1]; i--)
    if(!strcmp(local_var_stack[i].var_name, token))
        return local_var_stack[i].value;

/* в противном случае проверим, может быть это глобальная
переменная */
for(i=0; i < NUM_GLOBAL_VARS; i++)
    if(!strcmp(global_vars[i].var_name, s))
        return global_vars[i].value;

sntx_err(NOT_VAR); /* переменная не найдена */
return -1;
}

/* Если идентификатор является переменной, то
возвращается 1, иначе 0.
*/
int is_var(char *s)
{
    register int i;

    /* это локальная переменная? */
    for(i=lvartos-1; i >= call_stack[functos-1]; i--)
        if(!strcmp(local_var_stack[i].var_name, token))
            return 1;

    /* если нет - поиск среди глобальных переменных */
    for(i=0; i < NUM_GLOBAL_VARS; i++)
        if(!strcmp(global_vars[i].var_name, s))
            return 1;

    return 0;
}

/* Выполнение оператора if. */
void exec_if(void)
{
    int cond;

    eval_exp(&cond); /* вычисление if-выражения */

    if(cond) { /* истина - интерпретация if-предложения */
        interp_block();
    }
    else { /* в противном случае пропуск if-предложения
и выполнение else-предложения, если оно есть */
        find_eob(); /* поиск конца блока */
        get_token();

        if(tok != ELSE) {
            putback(); /* восстановление лексемы,
если else-предложение отсутствует */
            return;
        }
        interp_block();
    }
}

```

```

    }
}

/* Выполнение цикла while. */
void exec_while(void)
{
    int cond;
    char *temp;

    putback();
    temp = prog;
    /* запоминание адреса начала цикла while */
    get_token();
    eval_exp(&cond); /* вычисление управляющего выражения */
    if(cond) interp_block(); /* если оно истинно, то выполнить
                               интерпретацию */
    else { /* в противном случае цикл пропускается */
        find_eob();
        return;
    }
    prog = temp; /* возврат к началу цикла */
}

/* Выполнение цикла do. */
void exec_do(void)
{
    int cond;
    char *temp;

    putback();
    temp = prog; /* запоминание адреса начала цикла */

    get_token(); /* найти начало цикла */
    interp_block(); /* интерпретация цикла */
    get_token();
    if(tok != WHILE) sintx_err(WHILE_EXPECTED);
    eval_exp(&cond); /* проверка условия цикла */
    if(cond) prog = temp; /* если условие истинно, то цикл выполняется,
                           в противном случае происходит выход из цикла */
}

/* Поиск конца блока. */
void find_eob(void)
{
    int brace;

    get_token();
    brace = 1;
    do {
        get_token();
        if(*token == '{') brace++;
        else if(*token == '}') brace--;
    } while(brace);
}

/* Выполнение цикла for. */
void exec_for(void)
{

```

```

int cond;
char *temp, *temp2;
int brace ;

get_token();
eval_exp(&cond); /* инициализирующее выражение */
if(*token != ';') syntax_err(SEMI_EXPECTED);
prog++; /* пропуск ; */
temp = prog;
for(;;) {
    eval_exp(&cond); /* проверка условия */
    if(*token != ';') syntax_err(SEMI_EXPECTED);
    prog++; /* пропуск ; */
    temp2 = prog;

    /* поиск начала тела цикла */
    brace = 1;
    while(brace) {
        get_token();
        if(*token == '(') brace++;
        if(*token == ')') brace--;
    }

    if(cond) interp_block(); /* если условие выполнено,
                               то выполнить интерпретацию */
    else { /* в противном случае обойти цикл */
        find_eob();
        return;
    }
    prog = temp2;
    eval_exp(&cond); /* выполнение инкремента */
    prog = temp; /* возврат в начало цикла */
}
}

```

Обработка локальных переменных

Когда интерпретатор встречает зарезервированные слова `int` или `char`, он вызывает функцию `decl_local()`, которая размещает локальные переменные. Как отмечалось ранее, при выполнении программы интерпретатор не может встретить объявление глобальной переменной, потому что выполняется только код программы, записанный внутри функций. Следовательно, если встретилось объявление переменной, то это локальная переменная (или параметр — этот случай будет рассмотрен в следующем разделе). В структурированных языках локальные переменные хранятся в стеке. Если программа компилируется, то обычно используется системный стек. Однако при интерпретации стек с локальными переменными должен быть создан самим интерпретатором. В интерпретаторе Little C стек для локальных переменных хранится в массиве `local_var_stack`. Каждый раз, когда встречается локальная переменная, ее имя, тип и значение (первоначально равное нулю) заносятся в стек при помощи функции `local_push()`. Глобальная переменная `lvartos` является указателем стека. (Соответствующей функции извлечения из стека нет. Вместо этого стек локальных переменных переустанавливается каждый раз при возврате управления из функции `local_push()`. Зачем это сделано, будет видно из дальнейшего изложения.) Функции `decl_local()` и `local_push()` приведены ниже:

```

/* Объявление локальной переменной. */
void decl_local(void)
{
    struct var_type i;

    get_token(); /* определение типа */

    i.v_type = tok;
    i.value = 0; /* инициализация нулем */

    do { /* обработка списка */
        get_token(); /* определение имени переменной */
        strcpy(i.var_name, token);
        local_push(i);
        get_token();
    } while(*token == ',');
    if(*token != ';') syntax_err(SEMI_EXPECTED);
}

/* Затолкнуть локальную переменную в стек. */
void local_push(struct var_type i)
{
    if(lvartos > NUM_LOCAL_VARS)
        syntax_err(TOO_MANY_LVARS);

    local_var_stack[lvartos] = i;
    lvartos++;
}

```

Функция `decl_local()` сначала считывает тип объявленной переменной (или переменных) и инициализирует переменные нулями. Затем в цикле считывается список идентификаторов, разделенных запятыми. При каждой итерации цикла информация об очередной переменной заносится в стек локальных переменных. В конце функции `decl_local()` проверяется, является ли последняя лексема точкой с запятой.

Вызов функций, определенных пользователем

Выполнение функций, определенных пользователем, — это, наверное, самая трудная часть в реализации интерпретатора С. Интерпретатор должен начать чтение исходного текста программы с нового места, а затем вернуться в вызывающую процедуру после выхода из функции. Кроме того, он должен выполнить следующие три задачи: передать аргументы, разместить в памяти параметры и вернуть значение функции.

Все вызовы функций (кроме `main()`) осуществляются в синтаксическом анализаторе выражений из функции `atom()` с помощью вызова функции `call()`. Именно функция `call()` выполняет все необходимые при вызове функций действия. Текст функции `call()` вместе с ее вспомогательными функциями приведен ниже. Рассмотрим эту функцию подробнее:

```

/* Вызов функции. */
void call(void)
{
    char *loc, *temp;
    int lvartemp;

    loc = find_func(token);
    /* поиск точки входа функции */
    if(loc == NULL)

```

```

    sntx_err(FUNC_UNDEF); /* функция не определена */
else {
    lvartemp = lvartos; /* запоминание индекса стека локальных
                          переменных */
    get_args(); /* получение аргументов функции */
    temp = prog; /* запоминание адреса возврата */
    func_push(lvartemp); /* запоминание индекса стека локальных
                          переменных */
    prog = loc; /* установить prog в начало функции */
    get_params(); /* загрузка параметров функции
                  значениями аргументов */
    interp_block(); /* интерпретация функции */
    prog = temp; /* восстановление prog */
    lvartos = func_pop();
    /* восстановление стека локальных переменных */
}
}

/* Занесение аргументов функции
   в стек локальных переменных. */
void get_args(void)
{
    int value, count, temp[NUM_PARAMS];
    struct var_type i;

    count = 0;
    get_token();
    if(*token != '(') sntx_err(PAREN_EXPECTED);

    /* обработка списка значений */
    do {
        eval_exp(&value);
        temp[count] = value; /* временное запоминание */
        get_token();
        count++;
    } while(*token == ',');
    count--;
    /* теперь записать в стек local_var_stack в обратном порядке */
    for(; count >= 0; count--) {
        i.value = temp[count];
        i.v_type = ARG;
        local_push(i);
    }
}

/* Получение параметров функции. */
void get_params(void)
{
    struct var_type *p;
    int i;

    i = lvartos-1;
    do { /* обработка списка параметров */
        get_token();
        p = &local_var_stack[i];
        if(*token != ')') {
            if(tok != INT && tok != CHAR)
                sntx_err(TYPE_EXPECTED);

```



```

        p->v_type = token_type;
        get_token();

        /* связывание имени параметра с аргументом,
           который уже находится в стеке локальных переменных */
        strcpy(p->var_name, token);
        get_token();
        i--;
    }
    else break;
} while(*token == ',');
if(*token != '\0') sintx_err(PAREN_EXPECTED);
}

```

В первую очередь с помощью вызова функции `find_func()` функция `call()` находит адрес точки входа вызываемой функции в исходном тексте программы. Затем эта функция сохраняет текущее значение `lvartos` индекса стека локальных переменных в переменной `lvartemp`. Потом она вызывает функцию `get_args()`, которая обрабатывает все аргументы функции. Функция `get_args()` считывает список выражений, разделенных запятыми, и заносит их в стек локальных переменных в обратном порядке. (Обратный порядок занесения переменных применяется потому, что так их легче сопоставлять с соответствующими параметрами.) Значения переменных, записанные в стек, не имеют имен (стек — это всего лишь массив). Имена параметров даются им функцией `get_params()`, которая будет рассмотрена далее.

После обработки аргументов функции текущее значение указателя `prog` сохраняется в `temp`. Эта переменная указывает на точку возврата функции. После этого значение `lvartemp` заносится в стек вызовов функций. Доступ к этому стеку осуществляется с помощью функций `func_push()` и `func_pop()`. В данный стек при каждом вызове функции записывается значение `lvartos`. Значение `lvartos` представляет собой начальную точку в стеке локальных переменных для переменных (и параметров) вызванной функции. Значение на вершине стека вызовов функций используется для предотвращения доступа функции к переменным, которые в ней не объявлены.

Следующие две строки функции `call()` устанавливают указатель программы на начало функции и затем, вызывая функцию `get_params()`, устанавливают соответствие между формальными параметрами и значениями аргументов функции, которые уже находятся в стеке локальных переменных. Фактическое выполнение функции осуществляется вызовом `interp_block()`. После возврата управления из `interp_block()` указатель программы `prog` переустанавливается; он будет указывать на точку возврата, а индекс стека локальных переменных получит значение, которое он имел до вызова функции. На этом последнем шаге из стека фактически удаляются все локальные переменные функции.

Если вызванная функция содержит оператор `return`, то `interp_block()` перед возвратом в `call()` вызывает `func_ret()`, которая вычисляет возвращаемое значение; код этой функции приведен ниже:

```

/* Возврат из функции. */
void func_ret(void)
{
    int value;

    value = 0;
    /* вычисление возвращаемого значения, если оно есть */
    eval_exp(&value);

    ret_value = value;
}

```

Глобальная целочисленная переменная `ret_value` содержит возвращаемое функцией значение. На первый взгляд может показаться странным то, что локальной переменной `value` сначала присваивается возвращаемое значение функции, а затем это значение присваивается переменной `ret_value`. Причина здесь в том, что функции могут быть рекурсивными и функция `eval_exp()` для вычисления возвращаемого значения может вызвать ту же функцию.

Присваивание значений переменным

Возвратимся ненадолго к синтаксическому анализатору выражений. Когда встречается оператор присваивания, то сначала вычисляется значение выражения в правой части, а затем это значение присваивается переменной в левой части путем вызова `assign_var()`. Однако язык C структурирован и поддерживает глобальные и локальные переменные. Как же тогда в следующей программе

```
int count;

int main()
{
    int count, i;

    count = 100;

    i = f();

    return 0;
}

int f()
{
    int count;
    count = 99;
    return count;
}
```

функция `assign_var()` знает, какой именно переменной `count` нужно присвоить значение? Ответ на это простой: во-первых, локальные переменные имеют приоритет над одноименными глобальными, а, во-вторых, локальные переменные недоступны за пределами своих функций. Проанализируем, как применяются эти правила для разрешения коллизий в приведенных выше примерах операторов присваивания. Для этого рассмотрим функцию `assign_var()`:

```
/* Присваивание значения переменной. */
void assign_var(char *var_name, int value)
{
    register int i;

    /* сначала нужно узнать: это локальная переменная? */
    for(i=lvartos-1; i >= call_stack[functos-1]; i--) {
        if(!strcmp(local_var_stack[i].var_name, var_name)) {
            local_var_stack[i].value = value;
            return;
        }
    }
    if(i < call_stack[functos-1])
        /* Если это не локальная переменная, попытаемся найти ее в таблице
           глобальных переменных */

```

```

        for(i=0; i < NUM_GLOBAL_VARS; i++)
            if(!strcmp(global_vars[i].var_name, var_name)) {
                global_vars[i].value = value;
                return;
            }
    }
    syntax_err(NOT_VAR); /* переменная не найдена */
}

```

Как указывалось в предыдущем разделе, при каждом вызове функции индекс стека локальных переменных (*lvartos*) записывается в стек вызова функции. Это значит, что любая локальная переменная (или параметр), определенные в функции, будут записаны в стек выше точки, на которую указывает *lvartos*. Следовательно, функция *assign_var()* просматривает *local_var_stack*, начиная с текущего значения на верхушке стека, причем просмотр прекращается, когда индекс достигает того значения, которое было занесено при последнем вызове функции. Благодаря этому просматриваются только те переменные, которые являются локальными для данной функции. (Это также помогает вызывать рекурсивные функции, потому что текущее значение *lvartos* сохраняется при каждом вызове функции.) Таким образом, если в *main()* присутствует строка “*count = 100;*”, то *assign_var()* находит локальную переменную *count* внутри *main()*. В *f()* функция *assign_var()* находит переменную *count*, определенную в *f()*, а не в *main()*.

Если имя переменной не совпадает ни с одним из имен локальных переменных, то просматривается список глобальных переменных.

Выполнение оператора *if*

Итак, базовая структура интерпретатора Little C создана. Теперь к ней можно добавить некоторые управляющие операторы. Каждый раз, когда функция *interp_block()* встречает оператор с зарезервированным словом, она вызывает соответствующую функцию, обрабатывающую этот оператор. Один из самых легких операторов — *if*. Он обрабатывается функцией *exec_if()*:

```

/* Выполнение оператора if. */
void exec_if(void)
{
    int cond;

    eval_exp(&cond); /* вычисление выражения */

    if(cond) { /* если условие выполнено, то интерпретировать
                if-предложение */
        interp_block();
    }
    else { /* в противном случае пропустить if-предложение
            и выполнить else-предложение, если оно есть */
        find_eob(); /* поиск конца блока */
        get_token();

        if(tok != ELSE) {
            putback(); /* восстановление лексемы,
                        если else-предложение отсутствует */
            return;
        }
        interp_block();
    }
}

```

Рассмотрим эту функцию подробнее. В первую очередь она вызывает `eval_exp()` для вычисления значения условного выражения. Если условие (`cond`) выполнено (т.е. выражение имеет ненулевое значение), то функция вызывает рекурсивно `interp_block()`, выполняя тем самым блок `if`. Если `cond` ложно, вызывается функция `find_eob()`, которая передвигает указатель программы на оператор, следующий после блока `if`. Если там присутствует `else`-предложение, то оно обрабатывается функцией `exec_if()` и выполняется блок `else`. В противном случае выполняется следующий оператор программы.

Если блок `else` присутствует и выполняется блок `if`, то после его выполнения нужно каким-то образом обойти блок `else`. Эта задача выполняется в функции `interp_block()` путем вызова функции `find_eob()`, которая обходит блок после `else`. Помните, что в синтаксически правильной программе `else` обрабатывается функцией `interp_block()` только в одном случае — после выполнения блока `if`. Если выполняется блок `else`, то оператор `else` обрабатывается функцией `exec_if()`.

Обработка цикла `while`

Интерпретировать цикл `while`, как и `if`, довольно легко. Ниже приведен текст функции `exec_while()`, которая интерпретирует `while`:

```
/* Выполнение цикла while. */
void exec_while(void)
{
    int cond;
    char *temp;

    putback();
    temp = prog; /* запомнить адрес начала цикла while */
    get_token();
    eval_exp(&cond); /* проверка управляющего выражения */
    if(cond) interp_block();
    /* если оно истинно, то интерпретировать */
    else { /* в противном случае цикл пропускается */
        find_eob();
        return;
    }
    prog = temp; /* возврат к началу цикла */
}
```

Функция `exec_while()` работает следующим образом. Сначала лексема `while` возвращается во входной поток, а ее адрес сохраняется в переменной `temp`. Этот адрес будет использован интерпретатором, чтобы возвратиться к началу цикла (т.е. начать следующую итерацию с начала цикла `while`). Далее лексема `while` считывается заново для того, чтобы удалить ее из входного потока. После этого вызывается функция `eval_exp()`, которая вычисляет значение условного выражения цикла `while`. Если условие выполнено (т.е. условное выражение принимает значение ИСТИНА), то рекурсивно вызывается функция `interp_block()`, которая интерпретирует блок `while`. После возврата управления из `interp_block()` программный указатель `prog` устанавливается на начало цикла `while` и управление передается функции `interp_block()`, в которой весь процесс повторяется. Если условное выражение оператора `while` принимает значение ЛОЖЬ, то происходит поиск конца блока `while`, а затем выход из функции `exec_while()`.

Обработка цикла do-while

Обработка цикла do-while во многом похожа на обработку цикла while. Когда функция `interp_block()` встречает оператор `do`, она вызывает функцию `exec_do()`, исходный текст которой приведен ниже:

```
/* Выполнение цикла do. */
void exec_do(void)
{
    int cond;
    char *temp;

    putback();
    temp = prog; /* запомнить адрес начала цикла */

    get_token(); /* прочитать начало цикла */
    interp_block(); /* интерпретировать цикл */
    get_token();
    if(tok != WHILE) syntax_err(WHILE_EXPECTED);
    eval_exp(&cond); /* проверка условия цикла */
    if(cond) prog = temp; /* если условие выполнено, то цикл выполняется;
                          в противном случае происходит выход из цикла */
}
```

Главное отличие цикла do-while от цикла while состоит в том, что блок do-while выполняется всегда как минимум один раз, потому что его условное выражение находится после тела цикла. Поэтому `exec_do` сначала запоминает адрес начала цикла в переменной `temp`, а затем рекурсивно вызывает `interp_block()`, которая интерпретирует блок, ассоциированный с циклом. После возврата управления из `interp_block()` идет поиск соответствующего слова `while` и вычисляется значение условного выражения. Если условие выполнено, то `prog` устанавливается так, что его значение указывает на начало цикла, в противном случае выполнение продолжается со следующего оператора.

Цикл for

Интерпретация цикла `for` — задача значительно более трудная, чем интерпретация других операторов. Частично это объясняется тем, что структура цикла `for` в языке C была задумана в предположении компиляции программы. Главная трудность заключается в том, что условное выражение `for` должно проверяться в начале цикла, а часть приращения должна быть выполнена в его конце. Таким образом, эти две части цикла `for` в тексте исходной программы находятся рядом, а их интерпретация разделена выполнением блока цикла. Однако, после некоторой дополнительной работы, цикл `for` все же может быть интерпретирован.

Когда `interp_block()` встречает оператор `for`, вызывается функция `exec_for()`, текст которой приведен ниже:

```
/* Выполнение цикла for. */
void exec_for(void)
{
    int cond;
    char *temp, *temp2;
    int brace ;

    get_token();
    eval_exp(&cond); /* инициализирующее выражение */
}
```

```

if(*token != ';') sntx_err(SEMI_EXPECTED);
prog++; /* пропуск ; */
temp = prog;
for(;;) {
    eval_exp(&cond); /* проверка условия */
    if(*token != ';') sntx_err(SEMI_EXPECTED);
    prog++; /* пропуск ; */
    temp2 = prog;

    /* поиск начала тела цикла */
    brace = 1;
    while(brace) {
        get_token();
        if(*token == '(') brace++;
        if(*token == ')') brace--;
    }

    if(cond) interp_block(); /* если условие выполнено, то тело
                               цикла интерпретируется */
    else { /* в противном случае цикл пропускается */
        find_eob();
        return;
    }
    prog = temp2;
    eval_exp(&cond); /* выполнение инкремента */
    prog = temp; /* возврат к началу цикла */
}
}

```

Сначала функция обрабатывает инициализирующее выражение цикла `for`. Часть инициализации `for` выполняется только один раз; эта часть не подвергается циклической обработке. Затем указатель программы устанавливается так, чтобы он указывал на символ, следующий сразу после той точки с запятой, которой заканчивается часть инициализации. Наконец, после этого значение указателя присваивается переменной `temp`. А затем организовывается цикл, в котором проверяется условная часть цикла `for` и переменной `temp2` присваивается адрес начала части приращения. Далее производится поиск начала тела цикла и его (тела) интерпретация, если условное выражение принимает значение ИСТИНА. (В противном случае производится поиск конца тела цикла и выполнение продолжается с оператора, следующего после цикла `for`.) После рекурсивного вызова `interp_block()` выполняется часть приращения, после чего весь процесс повторяется.



Библиотечные функции Little C

Программы C, выполняемые интерпретатором Little C, никогда не компилируются и не компоуются, поэтому любая используемая в них библиотечная функция должна быть обработана непосредственно интерпретатором Little C. Лучше всего для этого создать интерфейсную функцию, вызываемую интерпретатором каждый раз при встрече библиотечной функции. Интерфейсная функция осуществляет подготовку к вызову библиотечной функции и обрабатывает возвращаемые значения.

В связи с ограниченным размером интерпретатора, Little C содержит только пять “библиотечных” функций: `getche()`, `putch()`, `puts()`, `print()` и `getnum()`. Конечно, в Стандарт C входит только функция `puts()`, выводящая строку на экран. Функция `getche()`, хотя и не предусмотрена в Стандарте, обычно включается в со-

став библиотек, так как она используется при работе в интерактивной среде. Она ожидает нажатия клавиши и возвращает связанное с ней значение. Следует отметить, что эта функция предусмотрена во многих компиляторах. Функция `putch()` также определена во многих компиляторах, предназначенных для создания программ, работающих в интерактивной среде. Она выводит на консоль один символ — ее аргумент. Вывод не буферизован. Функции `getnum()` и `print()` созданы автором. Функция `getnum()` возвращает целое число, равное числу, введенному с клавиатуры. Функция `print()` может выводить на экран как строковый, так и целочисленный аргумент, это очень удобно. Прототипы этих библиотечных функций приведены ниже¹:

```
int getche(void); /* считывание символа с клавиатуры
и возврат его значения */
int putch(char ch); /* вывод символа на экран */
int puts(char *s); /* вывод строки на экран */
int getnum(void); /* чтение целого числа с клавиатуры
и возврат его значения */
int print(char *s); /* вывод строки на экран */
или
int print(int i); /* вывод целого числа на экран */
```

Тексты процедур библиотеки функций Little C приведены ниже. Файл называется **LCLIB.C**.

```
/****** Библиотека функций Little C *****/

/* Сюда можно добавлять новые функции. */

#include <conio.h> /* если компилятор не поддерживает
данный заголовочный файл,
этот #include можно удалить */
#include <stdio.h>
#include <stdlib.h>

extern char *prog; /* указывает на текущий символ в программе */
extern char token[80]; /* содержит строковое представление лексемы */
extern char token_type; /* содержит тип лексемы */
extern char tok; /* содержит внутреннее представление лексемы */

enum tok_types {DELIMITER, IDENTIFIER, NUMBER, KEYWORD,
TEMP, STRING, BLOCK};

/* Эти константы используются для вызова sntx_err()
при синтаксической ошибке. Их список можно расширить.
ВНИМАНИЕ: SYNTAX обозначает
нераспознанную ошибку.
*/
enum error_msg
{SYNTAX, UNBAL_PARENS, NO_EXP, EQUALS_EXPECTED,
NOT_VAR, PARAM_ERR, SEMI_EXPECTED,
UNBAL_BRACES, FUNC_UNDEF, TYPE_EXPECTED,
NEST_FUNC, RET_NOCALL, PAREN_EXPECTED,
WHILE_EXPECTED, QUOTE_EXPECTED, NOT_STRING,
TOO_MANY_LVARS, DIV_BY_ZERO};

int get_token(void);
void sntx_err(int error), eval_exp(int *result);
```

¹ Язык Little C не поддерживает прототипы функций. Поэтому включать их в программу не следует. Здесь прототипы приведены в качестве справочной информации. — *Прим. перев.*

```

void putback(void);

/* Считывание символа с консоли. Если компилятор
   не поддерживает _getche(), то следует
   использовать getchar(). */
int call_getche()
{
    char ch;
    ch = _getche();
    while(*prog!='\n') prog++;
    prog++; /* продвижение к концу строки */
    return ch;
}

/* Вывод символа на экран. */
int call_putch()
{
    int value;

    eval_exp(&value);
    printf("%c", value);
    return value;
}

/* Вызов функции puts(). */
int call_puts(void)
{
    get_token();
    if(*token!='(') sntx_err(PAREN_EXPECTED);
    get_token();
    if(token_type!=STRING) sntx_err(QUOTE_EXPECTED);
    puts(token);
    get_token();
    if(*token!='\n') sntx_err(PAREN_EXPECTED);
    get_token();
    if(*token!=';') sntx_err(SEMI_EXPECTED);
    putback();
    return 0;
}

/* Встроенная функция консольного вывода. */
int print(void)
{
    int i;

    get_token();
    if(*token!='(') sntx_err(PAREN_EXPECTED);

    get_token();
    if(token_type==STRING) { /* вывод строки */
        printf("%s ", token);
    }
    else { /* вывод числа */
        putback();
        eval_exp(&i);
        printf("%d ", i);
    }

    get_token();

```



```

    if(*token!='') sntx_err(PAREN_EXPECTED);

    get_token();
    if(*token!=';') sntx_err(SEMI_EXPECTED);
    putback();
    return 0;
}

/* Считывание целого числа с клавиатуры. */
int getnum(void)
{
    char s[80];

    gets(s);
    while(*prog != ' ') prog++;
    prog++; /* продвижение к концу строки */
    return atoi(s);
}

```

Для того чтобы добавить в библиотеку новые функции, следует сначала включить их имена и адреса интерфейсных функций в массив `intern_func`. После этого необходимо создать соответствующие интерфейсные функции, используя приведенные выше функции как пример.

Компиляция и компоновка интерпретатора Little C

Три файла, составляющие интерпретатор Little C, должны быть откомпилированы и скомпонованы совместно. Для этого можно использовать любой современный компилятор C, включая Visual C++. Если используется Visual C++, то последовательность команд вызова компилятора выглядит следующим образом:

```

cl -c parser.c
cl -c lclib.c
cl littlec.c parser.obj lclib.obj

```

В некоторых версиях Visual C++ интерпретатору Little C может не хватить памяти для стека. В этом случае для увеличения стека нужно использовать опцию /F. Память, выделенная опцией /F6000, в большинстве случаев будет достаточной. Однако при интерпретации программ с глубоко вложенными рекурсивными вызовами эта память может оказаться все же недостаточной.

При использовании других компиляторов C нужно руководствоваться прилагаемыми к ним инструкциями.

Демонстрация Little C

Работа интерпретатора Little C демонстрируется с помощью следующих примеров программ.

Программа №1 иллюстрирует все средства программирования, поддерживаемые в Little C:

```

/* Little C. Демонстрационная программа №1.

```

```

    Эта программа демонстрирует работу всех средств

```

```

/* языка C, поддерживаемых интерпретатором Little C.
*/

int i, j; /* глобальные переменные */
char ch;

int main()
{
    int i, j; /* локальные переменные */

    puts("Программа демонстрации Little C.");

    print_alpha();

    do {
        puts("Введите число (0, если выход): ");
        i = getnum();
        if(i < 0 ) {
            puts("числа должны быть положительными, введите еще");
        }
        else {
            for(j = 0; j < i; j=j+1) {
                print(j);
                print("сумма равна");
                print(sum(j));
                puts("");
            }
        }
    } while(i!=0);

    return 0;
}

/* Сумма чисел от 0 до введенного числа. */
int sum(int num)
{
    int running_sum;

    running_sum = 0;

    while(num) {
        running_sum = running_sum + num;
        num = num - 1;
    }
    return running_sum;
}

/* Вывод на экран английского алфавита. */
int print_alpha()
{
    for(ch = 'A'; ch<='Z'; ch = ch + 1) {
        putchar(ch);
    }
    puts("");

    return 0;
}

```

Следующий пример демонстрирует использование вложенных циклов:

```
/* Пример с вложенными циклами. */
int main()
{
    int i, j, k;

    for(i = 0; i < 5; i = i + 1) {
        for(j = 0; j < 3; j = j + 1) {
            for(k = 3; k ; k = k - 1) {
                print(i);
                print(j);
                print(k);
                puts("");
            }
        }
    }
    puts("выполнено");

    return 0;
}
```

Следующая программа демонстрирует работу оператора присваивания:

```
/* Присваивание как операция. */
int main()
{
    int a, b;

    a = b = 10;

    print(a); print(b);

    while(a=a-1) {
        print(a);
        do {
            print(b);
        } while((b=b-1) > -10);
    }

    return 0;
}
```

Следующая программа демонстрирует выполнение рекурсивных функций. В ней функция `factr()` вычисляет факториал числа.

```
/* Демонстрация рекурсивных функций. */

/* возвращает факториал числа i */
int factr(int i)
{
    if(i<2) {
        return 1;
    }
    else {
        return i * factr(i-1);
    }
}

int main()
```

```

{
    print("Факториал от 4 равен: ");
    print(factr(4));

    return 0;
}

```

В следующей программе демонстрируются различные приемы использования аргументов функций:

```

/* Использование аргументов функций. */

```

```

int f1(int a, int b)
{
    int count;

    print("в функции f1");

    count = a;
    do {
        print(count);
    } while(count=count-1);

    print(a); print(b);
    print(a*b);
    return a*b;
}

int f2(int a, int x, int y)
{
    print(a); print(x);
    print(x / a);
    print(y*x);

    return 0;
}

int main()
{
    f2(10, f1(10, 20), 99);

    return 0;
}

```

И, наконец, в последнем примере демонстрируется работа операторов цикла:

```

/* Операторы цикла. */
int main()
{
    int a;
    char ch;

    /* цикл while */
    puts("Введите число: ");
    a = getnum();
    while(a) {
        print(a);
        print(a*a);
        puts("");
        a = a - 1;
    }
}

```

```

    }

    /* цикл do-while */
    puts("Введите символ, если выход, то 'q' ");
    do {
        ch = getche();
    } while(ch != 'q');

    /* цикл for */
    for(a=0; a<10; a = a + 1) {
        print(a);
    }

    return 0;
}

```

Усовершенствование интерпретатора Little C

Рассмотренный в этой главе интерпретатор Little C разрабатывался таким образом, чтобы его принцип действия был, по возможности, очевидным. При разработке интерпретатора преследовалась цель сделать его максимально легким для понимания. Другой целью было сделать его легко расширяемым. Поскольку преследовались именно эти цели, интерпретатор Little C не обладает значительным быстродействием или эффективностью. Однако базовая структура интерпретатора корректна, а скорость выполнения программ можно увеличить, пользуясь указаниями, приведенными в этом разделе.

Фактически во всех коммерческих интерпретаторах роль программы предварительного прохода значительно шире, чем в Little C. Интерпретируемый исходный текст программы преобразуется из формы ASCII, которая удобна программисту для чтения, во внутреннюю форму. В этой внутренней форме все, кроме заключенных в двойные кавычки строк и констант, преобразуется в лексемы, состоящие из одного числа, аналогично тому, как это делает Little C для зарезервированных слов. При работе интерпретатора Little C довольно часто сравниваются строки. Например, всякий раз при поиске переменной или функции выполняется несколько сравнений строк. Процедура сравнения строк занимает много времени, что конечно же значительно снижает быстродействие программы. Но если каждую лексему исходной программы преобразовать в целое число, то можно использовать намного более быстродействующую операцию сравнения целых чисел. Преобразование исходного текста программы во внутреннюю форму — *единственное наиболее существенное изменение*, повышающее эффективность Little C. Благодаря этому преобразованию повышение скорости будет весьма ощутимым.

Другой способ улучшения, полезный главным образом для интерпретации больших программ, — создание специальных процедур для поиска переменных и функций. Применяющийся метод поиска основан на последовательном просмотре имен переменных и программ. Такой просмотр занимает много времени, даже если имена переменных и программ преобразованы в целочисленные лексемы. Однако можно заменить этот метод поиска другим, более быстрым методом, используя, например, двоичное дерево или один из методов хэширования.

Как указывалось ранее, одним из ограничений Little C по сравнению с грамматикой полного C является требование заключать объекты некоторых операторов, таких как `if`, в фигурные скобки независимо от того, единственный это оператор или блок операторов. Это предусмотрено с целью существенного упрощения функции `find_eob()`, которая ищет конец блока после выполнения одного из управляющих операторов. Функция `find_eob()` попросту ищет закрывающуюся скобку, соответствующую скобке, открывающей блок. Устранение этого ограничения будет интересным

упражнением для читателя. Для этого можно, например, усовершенствовать функцию `find_eob()` таким образом, чтобы она искала конец оператора, выражения или блока. Однако следует иметь в виду, что для операторов `if`, `while`, `do-while` и `for` потребуются различные подходы, если в них используется единственный оператор.

Расширение Little C

Расширять возможности интерпретатора Little C можно в двух направлениях: добавлять в него новые средства языка C и дополнительные средства программирования. Эти усовершенствования кратко рассматриваются в следующих разделах.

Добавление новых средств в язык Little C

Существует две категории операторов C, которые можно включить в Little C. В первую категорию входят дополнительные выполняемые операторы C, такие как `switch`, `goto`, `break` и `continue`. Если предыдущий материал изучен достаточно тщательно, то их добавление в Little C не составит большого труда.

Во вторую категорию входит поддержка новых типов данных. В интерпретаторе Little C для этого есть некоторые “зацепки”. Например, в структуре `var_type` есть поле для типов переменных. Для включения дополнительных базовых типов (например, `float`, `double` или `long`) нужно просто увеличить размер поля до размера наибольшего элемента.

Учтите, что реализация указателей не труднее, чем реализация других типов данных. Однако для этого нужно будет добавить в синтаксический анализатор выражений поддержку операций для работы с указателями.

После реализации операций для работы с указателями легко добавить массивы. Память для массива следует выделять динамически, используя `malloc()`, а указатель на массив нужно хранить в поле `value` структуры `var_type`.

Более трудная задача — добавление структур и объединений. Проще всего это сделать, используя `malloc()` для выделения объекту памяти, причем указатель на объект нужно сохранить в поле `value` структуры `var_type`. (Для обработки передачи структур и объединений в качестве параметров нужно будет написать специальную программу.)

Для поддержки различных типов возвращаемых функциями значений нужно использовать поле `ret_type` структуры `func_type`. Это поле определяет тип возвращаемых функций данных. В текущей версии интерпретатора оно объявлено, но не используется.

Можно также добавить в Little C поддержку комментариев вида `//`. Это нетрудно сделать, изменив функцию `get_token()`.

И наконец, в интерпретатор Little C несложно добавить средства, не входящие в состав языка C. Это особенно увлекательное упражнение — заставить интерпретатор делать то, что в языке не предусмотрено. Например, можно добавить конструкцию языка Pascal `REPEAT-UNTIL`. Если при этом возникают трудности, как средство отладки можно использовать вывод каждой лексемы в процессе ее обработки.

Создание дополнительных средств программирования

Кроме средств языка, в интерпретатор несложно добавить также новые средства программирования. Например, можно добавить средства трассировки, выводящие на экран в процессе выполнения программы каждую лексему отдельно. Еще можно добавить возможность вывода значений переменных при выполнении программы, а также, например, встроенный редактор, который позволит редактировать и выполнять программу без перехода в автономный редактор.

Предметный указатель

#

##, оператор, 250
#, модификатор, 208
#, оператор, 250
#define, директива препроцессора, 242
#elif, директива препроцессора, 246
#else, директива препроцессора, 246
#endif, директива препроцессора, 246
#error, директива препроцессора, 244
#if, директива препроцессора, 245
#ifdef, директива препроцессора, 247
#ifndef, директива препроцессора, 248
#include, директива препроцессора, 245
#line, директива препроцессора, 249
#pragma, директива препроцессора, 250
#undef, директива препроцессора, 248

%

%%, спецификатор преобразования, 202; 209
%[], спецификатор преобразования, 209
%a, спецификатор преобразования, 202; 209
%c, спецификатор преобразования, 202; 209
%d, спецификатор преобразования, 202; 209
%e, спецификатор преобразования, 202; 209
%f, спецификатор преобразования, 202; 209
%g, спецификатор преобразования, 202; 209
%i, спецификатор преобразования, 202; 209
%n, спецификатор преобразования, 202; 209
%o, спецификатор преобразования, 202; 209
%p, спецификатор преобразования, 202; 209

%s, спецификатор преобразования, 202; 209
%u, спецификатор преобразования, 202; 209
%x, спецификатор преобразования, 202; 209

*

*, модификатор, 208

?

?, оператор, 87
?, специальная символьная константа, 62

\

\, специальная символьная константа, 62

—

__DATE__, макрос, 251
__FILE__, идентификатор, 249
__func__, идентификатор, 268
__LINE__, идентификатор, 249
__STDC__, макрос, 251
__STDC_HOSTED__, макрос, 251; 264
__STDC_IEC_559__, макрос, 264
__STDC_IEC_559_COMPLEX__, макрос, 264
__STDC_ISO_10646__, макрос, 264
__STDC_VERSION__, макрос, 251; 264
__TIME__, макрос, 251
_Bool, 259; 434
_Complex, 260; 426
_Exit(), 399
_getch(), функция, 198
_getche(), функция, 198
_Imaginary, 260; 426
_Pragma, 263

A

a, режим, 219
a, специальная символьная константа, 62
a+, режим, 220
a+b, режим, 220
ab, режим, 220
abort(), 392
abs(), 393
American National Standards Institute, 32
ANSI, 32
API, 279; 576
Application
 Program Interface, 279
 Programming Interface, 576
assert(), 393
assert.h, заголовок, 280
atexit(), 394
atof(), 394
atoi(), 395
atol(), 396
atoll(), 396
ATOM, 585

B

b, специальная символьная константа, 62
BCPL, 32
Binary tree, 484
Bitmap, 575
BKACK_BRUSH, 585
bool, 260; 580
break, оператор, 102
Brian Kernighan, 32
bsearch(), 397
btowc(), 424
Bubble sort, 440
BYTE, 580

C

Calling sequence, 607
calloc(), 386
Comparand, 448
complex, 260
complex.h, заголовок, 267; 280
const, ключевое слово, 262
continue, оператор, 104
CreateWindow(), 585
ctype.h, заголовок, 280
CW_USEDEFAULT, 586

D

defined, оператор, 249
Definition files, 589
Dennis Ritchie, 32
Dependent files, 597
DispatchMessage(), 588
div(), 398
DKGRAY_BRUSH, 585
DLL, 278
double_Complex, 260
double_Imaginary, 260
do-while, цикл, 100
DWORD, 580
Dynamic-Link Library, 278

E

EOF, макрос, 219
errno.h, заголовок, 280
exit(), функция, 103; 398
EXIT_FAILURE, 398
EXIT_SUCCESS, 398
Expression parsing, 510

F

f, специальная символьная константа, 62
false, 434
fclose(), функция, 218; 220; 285
fenv.h, заголовок, 267
feof(), функция, 223; 218; 286
ferror(), функция, 218; 226; 287
fflush(), функция, 218; 287
fgetc(), функция, 218; 221; 288
fgetpos, функция, 289
fgets(), функция, 199; 218; 224; 289
fgetws(), 420
float.h, заголовок, 280
float_Complex, 260
float_Imaginary, 260
fopen(), функция, 218; 219; 290
FOPEN_MAX, макрос, 219
for, оператор, 92
for, цикл, 97
fprintf(), функция, 218; 235; 291
fputc(), функция, 218; 221; 292
fputs(), функция, 218; 224; 293
fputwc(), 420
fputws(), 420
fread(), функция, 228; 293
free(), функция, 140; 387

freopen(), функция, 238; 294
fscanf(), функция, 218; 235; 295
fseek(), функция, 218; 234; 296
fsetpos, функция, 297
ftell(), функция, 218; 298
fwide(), 421
fwprintf(), 420
fwrite(), функция, 228; 298
fwscanf(), 420

G

GDI, 576
General problem solver, 530
getc(), функция, 218; 221; 299
getchar(), функция, 197; 300
getenv(), 399
gets(), функция, 199; 300
GetStockObject(), 585
getwc(), 420
getwchar(), 420
goto, оператор, 101
GPS, 530
Graphics Device Interface, 576

H

HANDLE, 580
Hashing, 503
hh, модификатор, 207; 267
Hoare Charles Antony Richard, 448
HOLLOW_BRUSH, 585
hPrevInst, 583
hThisInst, 583
HUGE_VAL, 346; 408
HUGE_VALL, 411
HWND, 580

I

IDC_ARROW, 585
IDC_CROSS, 585
IDC_HAND, 585
IDC_IBEAM, 585
IDC_WAIT, 585
IDE, 601
IDI_APPLICATION, 584
IDI_ERROR, 584
IDI_INFORMATION, 584
IDI_QUESTION, 584
IDI_WARNING, 584
IDI_WINLOGO, 584

if, оператор, 82
if-else-if, 85
imaginary, 260
inline, ключевое слово, 168; 609
int_fast32_t, расширенный тип, 270
int_least16_t, расширенный тип, 270
int16_t, расширенный тип, 270
Integrated development environment, 601
International Standards Organization, 32
intmax_t, расширенный тип, 270
isalnum, функция, 320
isalpha, функция, 321
isblank, функция, 322
isctrl, функция, 322
isdigit, функция, 323
isgraph, функция, 324
islower, функция, 324
ISO, 32
iso646.h, заголовок, 268
isprint, функция, 325
ispunct, функция, 325
isspace, функция, 326
isupper, функция, 327
isxdigit, функция, 327

K

Ken Thompson, 32

L

L, модификатор, 207
labs(), 400
ldiv(), 401
limits.h, заголовок, 280
Little C, 625
ll, модификатор, 207; 267
llabs(), 400
lldiv(), 401
LLONG_MAX, 411
LLONG_MIN, 411
LoadCursor(), 584
LoadIcon(), 584
LoadImage(), 584
locate.h, заголовок, 280
LONG, 580
long double_Complex, 260
long double_Imaginary, 260
long long int, 260
long, спецификатор типа, 45
LONG_MAX, 410
LONG_MIN, 410

longjmp(), 402
LPCSTR, 580
LPSTR, 580
lpszArgs, 583
LPVOID, 586
LRESULT, 579
LTGRAY_BRUSH, 585

M

MAKE, 597
MAKEFILE, 600
make-файл, 597
malloc, функция, 140; 387
Martin Richards, 32
math.h, заголовок, 280
MB_CUR_MAX, 415
mblen(), функция, 403
mbrlen(), функция, 424
mbrtowc(), функция, 424
mbsinit(), функция, 424
mbsrtowcs(), функция, 424
mbstowcs(), функция, 403
mbtowc(), функция, 404
memchr, функция, 328
memcmp, функция, 329
memcpy, функция, 330
memmove, функция, 331
memset, функция, 332
Microsoft C/C++, 597
MSG, 580

N

n, специальная символьная константа, 62
NMAKE, 597
Node, 484
NULL, макрос, 219; 392
nWinMode, 583

P

Pascal-соглашение о вызовах, 579
perror, функция, 301
printf, функция, 302
putc(), функция, 218; 221; 305
putchar(), функция, 197; 305
puts, функция, 306
putwc(), 420
putwchar(), 420

Q

qsort(), 404
quicksort, 404

R

r, режим, 219
r, специальная символьная константа, 62
r+, режим, 220
r+b, режим, 220
raise(), 405
rand(), 406
RAND_MAX, 406
rb, режим, 219
realloc(), 388
remove(), функция, 218; 227; 307
rename, функция, 307
restrict, ключевое слово, 262
return, оператор, 101
Returning sequence, 607
rewind(), функция, 218; 225; 308
Root, 484

S

scanf, функция, 208; 308
SEEK_CUR, макрос, 234
SEEK_END, макрос, 234
SEEK_SET, макрос, 234
setbuf, функция, 312
setjmp(), функция, 406
setjmp.h, заголовок, 280
setvbuf, функция, 312
Shaker sort, 443
Shell Donald Lewis, 446
short, спецификатор типа, 45
ShowWindow(), 586
SIG_ERR, 407
signal(), 407
signal.h, заголовок, 280
signed, спецификатор типа, 45
sizeof, 261
snprintf, функция, 313
Sparse array, 494
sprintf, функция, 313
srand(), 407
sscanf, функция, 314
stdarg.h, заголовок, 280
stdbool.h, заголовок, 268
STDC CX_LIMITED_RANGE, прагма, 264

STDC FENV_ACCESS, прагма, 264
 STDC FP_CONTRACT, прагма, 263
 stddef.h, заголовок, 280
 stderr, поток, 237
 stdin, поток, 237
 stdint.h, заголовок, 268
 stdio.h, заголовок, 280
 stdlib.h, заголовок, 280
 stdout, поток, 237
 strcat, функция, 111; 332
 strchr, функция, 111; 333
 strcmp, функция, 111; 333
 strcoll, функция, 334
 strcpy, функция, 111; 335
 strcspn, функция, 335
 strerror, функция, 336
 string.h, заголовок, 280
 strlen, функция, 111; 336
 strncat, функция, 336
 strncmp, функция, 337
 strncpy, функция, 338
 strpbrk, функция, 339
 strchr, функция, 339
 strspn, функция, 340
 strstr, функция, 111; 340
 strtod, функция, 408
 strtod, функция, 409
 strtok, функция, 341
 strtol, функция, 410
 strtold, функция, 411
 strtoll, функция, 411
 strtoul, функция, 411
 strtoull, функция, 412
 struct, ключевое слово, 170
 strxfrm, функция, 342
 Subtree, 484
 SW_HIDE, 587
 SW_MAXIMIZE, 587
 SW_MINIMIZE, 587
 SW_RESTORE, 587
 switch, оператор, 89
 swprintf, 420
 swscanf, 420
 system, 413

T

t, специальная символьная константа, 62
 Target files, 597
 Terminal node, 484
 tgmth.h, заголовок, 268
 time.h, заголовок, 280

tmpfile, функция, 315
 tmpnam, функция, 315
 Token, 512
 tolower, функция, 342
 toupper, функция, 343
 TranslateMessage(), 588
 Tree traversal, 485
 true, 434
 typedef, ключевое слово, 170

U

UINT, 580
 uintmax_t, расширенный тип, 270
 ULONG_MAX, 412
 ULONG_MAX, 412
 ungetc, функция, 316
 ungetwc(), 420
 union, ключевое слово, 184
 unsigned long long int, 260
 unsigned, спецификатор типа, 45

V

v, специальная символьная константа, 62
 va_arg(), 413
 va_copy(), 413
 va_end(), 413
 va_start(), 413
 Variable-length array, 261
 vfprintf, функция, 317
 vfscanf, функция, 318
 vfwprintf(), 421
 vfwscanf(), 421
 Visual C++, 601
 VLA, 261
 volatile, ключевое слово, 262
 vprintf, функция, 317
 vscanf, функция, 318
 vsnprintf, функция, 317
 vsprintf, функция, 317
 vsscanf, функция, 318
 vswprintf(), функция, 421
 vswscanf(), функция, 421
 vwprintf(), функция, 421
 vwscanf(), функция, 421

W

w, режим, 219
 w+, режим, 220
 w+b, режим, 220

- wb, режим, 219
- wctomb(), 424
- wscat(), 422
- wcschr(), 422
- wscmp(), 422
- wscoll(), 422
- wscpy(), 422
- wscspn(), 422
- wcsftime(), 423
- wcslen(), 422
- wscncat(), 422
- wscncmp(), 422
- wscncpy(), 422
- wcsprkr(), 422
- wcsrchr(), 422
- wcsrtombs(), 424
- wcsspn(), 422
- wcsstr(), 422
- wctod(), 423
- wcstof(), 423
- wctok(), 422
- wctol(), 423
- wctold(), 423
- wctoll(), 423
- wctombs(), 415
- wctoul(), 423
- wctoull(), 423
- wcsxfrm(), 422
- wctob(), 424
- wctomb(), 415
- wctype.h, заголовок, 268
- WEOF, 418
- while, цикл, 98
- WHITE_BRUSH, 585
- Win16, 576
- Win32, 577
- WinAPI-соглашение о вызовах, 579
- Window class, 579
- Window function, 579
- Window procedure, 579
- Windows 95, 577
- Windows 98, 577
- wmemchr(), 423
- wmemcmp(), 423
- wmemcpy(), 423
- wmemmove(), 423
- wmemset(), 423
- WNDCLASSEX, 580; 583
- WORD, 580
- wprintf(), 421
- WS_HSCROLL, 586
- WS_MAXIMIZEBOX, 586
- WS_MINIMIZEBOX, 586

- WS_OVERLAPPED, 586
- WS_OVERLAPPEDWINDOW, 586
- WS_SYSMENU, 586
- WS_VSCROLL, 586
- wscanf(), 421

А

- Абсолютный код, 277
- Агрегатный тип данных, 170
- Администратор оверлейной загрузки, 278
- Адресная арифметика, 129
- Алгоритм
 - быстрой сортировки quicksort, 404
 - наискорейшего подъема, 549
 - пузырьковой сортировки, количество обменов, 442
- Аргумент командной строки, 154
- Аргументы функции main, 154
- Арифметические операции
 - %, 66
 - *, 66
 - /, 66
 - +, 66
 - ++, 66
 - вычитание, 66
 - декремент, 66
 - деление, 66
 - инкремент, 66
 - остаток от деления, 66
 - сложение, 66
 - увеличение, 66
 - уменьшение, 66
 - умножение, 66
 - унарный минус, 66
- Ассемблерный код, 607

Б

- Бесконечный цикл, 96
- Библиотека, 41; 279
 - поддержки среды вычислений с плавающей точкой, 430
- Библиотечные файлы, 279
- Битмапка \i, 575
- Битовое поле, 170
- Блок
 - операторов, 105
 - управления файлом, 218
- Брайан Керниган, 32

В

Ввод

- адреса, 211
- одиночных символов, 210
- строк, 210
- целых значений без знака, 210
- чисел, 209

Вертикальная

- вертикальная табуляция,
- специальная символьная константа, 62

- двухтавровая балка, 585
- полоса прокрутки, 586

Вершина (графа), 531

Вершина дерева, 484

Вложенные

- директивы #include, 245
- операторы switch, 92
- условные операторы, 84

Внешняя ссылка, 276

Возврат каретки, специальная

- символьная константа, 62

Возвращающая последовательность, 607

Восклицательный знак, 584

Восходящий метод, 592

Восьмеричная константа, специальная

- символьная константа, 62

Восьмеричные константы, 61

Время

- выполнения, 41
- компиляции, 41

Встроенные прагмы, 263

Выбор структуры данных, 593

Вывод

- адреса, 204
- символов, 202
- чисел, 202

Вызывающая последовательность, 607

Высота, 484

Г

Глобальные переменные, 51

Горизонтальная

- полоса прокрутки, 586
- табуляция, специальная символьная константа, 62

Граничное значение для количества аргументов при вызове функции, 269

значащих символов во внешнем идентификаторе, 269

значащих символов во внутреннем идентификаторе, 268

уровней вложенности блоков, 268

уровней вложенности условных включений, 268

членов структуры или объединения, 269

Графический интерфейс устройств, 576

Д

Двоичное дерево, 484

Двоичный

- поиск, 458
- поток, 217

Двойная кавычка, специальная

- символьная константа, 62

Двусвязный список, 476

Двухмерный массив, 112

Деннис Ритчи, 32

Дерево

- вырожденное, 489
- сбалансированное, 489

Дескриптор, 580

- кисти, 585
- стандартных объектов отображения, 585

Диалоговое окно, 576

Диапазон значений

- типов float и double, 44
- получаемых сообщений, 588

Динамически подсоединяемые библиотеки, 278

Динамическое

- распределение, 140
- связывание, 278

Директива

- FENV_ACCESS, 430

Директива препроцессора, 242

- #define, 242

- #elif, 246

- #else, 246

- #endif, 246

- #error, 244

- #if, 245

- #ifdef, 247

- #ifndef, 248

- #include, 245

- #line, 249

#pragma, 250
#undef, 248
Директивы условной компиляции, 245
Дозапись, 218
 потока, 228
Доступ к членам структуры, 172

3

Зависимые файлы, 597
Заголовок, 166; 279
 assert.h, 280; 393
 complex.h, 267; 280; 426; 433
 ctype.h, 280
 errno.h, 280
 fenv.h, 267; 280; 430
 float.h, 280
 inttypes.h, 280; 432
 iso646.h, 268; 280
 limits.h, 280
 locate.h, 280
 math.h, 280; 426; 433
 setjmp.h, 280; 402; 406
 signal.h, 280; 405; 407
 stdarg.h, 280; 413
 stdbool.h, 268; 280; 434
 stddef.h, 280; 431; 432
 stdint.h, 268; 280
 stdio.h, 280
 stdlib.h, 280
 string.h, 280
 tgmath.h, 268; 280; 433
 time.h, 280
 wchar.h, 280; 418; 421; 422; 423; 424
 wctype.h, 268; 280; 418
Заголовок окна, 586
Заголовочный файл
 WINDOWS.H, 583
Заккрытие файла, 220
Запись символа, 221
Зарезервированные слова, 632
Знак вопроса, 584
 специальная символьная константа,
 62
Знаки пунктуации, 632

И

Идентификатор
 __FILE__, 249
 func, 268
 __LINE__, 249

Идентификаторы, 632
Инициализация
 безразмерных массивов, 120
 массива, 118
 переменных, 60
 указателя, 135
Инициализация
Инкрементное тестирование, 619
Интегрированная среда разработки,
601
 Visual C++, 601
Интерпретатор
 включение дополнительных базовых
 типов, 680
 вложенные циклы, 677
 встроенный редактор, 680
 вывод значений переменных при
 выполнении программы, 680
 вывод сообщения об ошибке, 636
 вызов функций, определенных
 пользователем, 665
 выполнение
 оператора if, 669
 рекурсивных функций, 677
 глобальная целочисленная
 переменная ret_value, 668
 демонстрация, 675
 рекурсивных функций, 677
 добавление
 новых средств, 680
 структур и объединений, 680
 дополнительные средства
 программирования, 680
 зарезервированные слова, 632
 знаки пунктуации, 632
 значение_переменной, 630
 идентификаторы, 632
 именующее выражение, 630
 компиляция и компоновка, 675
 локальные переменные, 627
 массив
 func_table, 650
 global_vars, 650
 массивы, 680
 нисходящий синтаксический
 анализатор, 636
 обработка
 локальных переменных, 664
 цикла do-while, 671
 цикла while, 670
 объявление локальной переменной,
 665
 ограничения языка Little C, 626

- оператор_сравнения, 630
- операторы цикла, 678
- определение языка Little C, 625
- опция /F, 675
- отличие цикла do-while от цикла while, 671
- память для стека, 675
- переменная целого типа gvar_index, 650
- повышение эффективности, 679
- поддержка
 - комментариев вида //, 680
 - новых типов данных, 680
- поиск
 - всех функций программы, 648
 - переменных и функций, 679
- практическое значение, 624
- предварительный проход, 648
- приемы использования аргументов функций, 678
- пример с вложенными циклами, 677
- присваивание
 - значений переменным, 668
 - как операция, 677
- прототипы
 - библиотечных функций, 673
 - функций, 627
- размещение глобальных переменных, 648
- расширение Little C, 680
- реализация указателей, 680
- синтаксический разбор исходного текста программы, 631
- средства трассировки, 680
- тексты процедур библиотеки функций Little C, 673
- указатель
 - p_buf, 635
 - prog, 635
- усовершенствование, 679
- факториал числа, 677
- флажок block, 651
- функция
 - assign_var(), 668
 - atom(), 646
 - call(), 665
 - decl_global(), 649
 - decl_local(), 665
 - exec_do(), 671
 - exec_while(), 670
 - factr(), 677
 - find_cob(), 679

- get_token(), 635
- getche(), 672
- getnum(), 673
- interp_block(), 651
- prescan(), 649
- print(), 673
- sntx_err(), 635
- цикл for, 671
- языка small BASIC, 628
- Интерфейс прикладного программирования, 576
- Инфиксная запись, 468
- Исполняемый файл, 276
- Исходный текст, 41
- Исчерпывающий поиск, 534

К

- Квалификатор типа
 - const, 53
 - volatile, 54
- Квалификатор типа restrict, 257
- Кен Томпсон, 32
- Класс окна, 579
- Ключ, 438
- Ключевое слово
 - const, 262
 - inline, 168; 258
 - restrict, 262
 - static, 261
 - struct, 170
 - typedef, 170
 - union, 184
 - volatile, 262
- Кнопка
 - развертывания, 586
 - свертывания, 586
- Комбинаторный взрыв, 534
- Комментарии, 251; 262
- Компаранд, 448
 - выбор, 450
- Компилятор, 36
- Компоновщик, 41
- Компоновщик оверлеев, 277
- Корень, 484
- Круглые скобки, 79
- Куча, 140

Л

Лексема, 512; 631
Лист, 484; 531
Логические значения, 82
Логическое значение
 False, 82
 True, 82
ИСТИНА, 82
ЛОЖЬ, 82
Логотип Windows, 584
Локальные переменные, 47

М

Макрос

__bool_true_false_are_defined, 434
__DATE__, 251
__STDC__, 251
__STDC_HOSTED__, 251; 264
__STDC_IEC_559__, 264
__STDC_IEC_559_COMPLEX__, 264
__STDC_ISO_10646__, 264
__STDC_VERSION__, 251; 264
__TIME__, 251
_Complex_I, 426
_Imaginary_I, 426
assert(), 393
bool, 434
complex, 426
EDOM, 346
EILSEQ, 424
EOF, 219
ERANGE, 346
EXIT_FAILURE, 392
EXIT_SUCCESS, 392
false, 434
FE_ALL_EXCEPT, 430
FE_DFL_ENV, 430
FE_DIVBYZERO, 430
FE_DOWNWARD, 430
FE_INEXACT, 430
FE_INVALID, 430
FE_OVERFLOW, 430
FE_TONEAREST, 430
FE_TOWARDZERO, 430
FE_UNDERFLOW, 430
FE_UPWARD, 430
FOPEN_MAX, 219
fpclassify, 346
HUGE_VAL, 346

HUGE_VALF, 346
HUGE_VALL, 346
I, 426
imaginary, 426
INFINITY, 346
int_fast32_t, 431
int16_t, 431
isfinite, 346
isgreater, 346
isgreaterequal, 346
isinf, 346
isless, 346
islessequal, 346
islessgreater, 346
isnan, 346
isnormal, 346
isunordered, 346
LC_ALL, 381
LC_COLLATE, 381
LC_CTYPE, 381
LC_MONETARY, 381
LC_NUMERIC, 381
LC_TIME, 381
MATH_ERREXCEPT, 346
math_errhandling, 346
MATH_ERRNO, 346
MB_CUR_MAX, 392
NaN, 346
NULL, 219; 392; 418
RAND_MAX, 392
SEEK_CUR, 234
SEEK_END, 234
SEEK_SET, 234
setjmp(), 406
SIG_DFL, 407
SIG_IGN, 407
SIGABRT, 406
SIGFPE, 406
SIGILL, 406
SIGINT, 406
signbit, 346
SIGSEGV, 406
SIGTERM, 406
true, 434
uint32_t, 431
va_arg, 413
va_copy(), 413
va_end, 413
va_start, 413
WCHAR_MAX, 418
WCHAR_MIN, 418
WEOF, 418
WS_OVERLAPPEDWINDOW, 586

Макросы

- вида `int_fastN_t`, 431
- вида `int_leastN_t`, 431
- вида `intN_t`, 431
- вида `uint_fastN_t`, 431
- вида `uint_leastN_t`, 431
- вида `uintN_t`, 431
- математические обобщенного типа, 433

Мартин Ричардс, 32

Массив, 108

- двухмерный, 112
- динамическое выделение памяти, 141
- инициализация, 118
- многомерный, 116
- одномерный, 108
- передача одномерного массива в функцию, 110
- переменной длины, 121; 261
- размер, 108
- с переменными границами, 266
- строк, 115
- указателей, 133

Машинный код, 36

Международная организация по стандартизации, 32

Метод

- вставки, 439
- выбора, 439
- обмена, 439
- поиска, 457
- половинного деления, 458
- рекурсивного спуска, 510
- сокрытия кода и данных, 595
- удаления вершин, 557
- удаления путей, 557

Минимально допустимый диапазон значений

- `char`, 45
- `double`, 45
- `float`, 45
- `int`, 45
- `long double`, 45
- `long int`, 45
- `long long int`, 45
- `short int`, 45
- `signed char`, 45
- `signed int`, 45
- `signed long int`, 45
- `signed short int`, 45
- `unsigned char`, 45
- `unsigned int`, 45

- `unsigned long int`, 45
- `unsigned long long int`, 45
- `unsigned short int`, 45

Многомерный массив, 116

Многоуровневая адресация, 134

Модель вызова функций `CALLBACK`, 579

Модификатор

- `#`, 208
- `*`, 208
- `hh`, 207; 267
- `L`, 207
- `ll`, 207; 267
- минимальной ширины поля, 205
- точности, 206

Модификаторы формата, 213

Н

Набор сканируемых символов, 211

Назначенные инициализаторы, 266

Наискорейший подъем, 549

Национальный институт стандартизации США, 32

Небуферизованная система ввода/вывода, 216

Неинициализированный указатель, 144

Неявные объявления функций, 270

Нисходящий

- метод, 592
- синтаксический анализатор, 631

Новая строка, специальная символьная константа, 62

О

Области видимости, 52

Область поиска, 531

Обмен, 440

Обратный

- обход, 486
- слэш, специальная символьная константа, 62

Обход

- в глубину, 486
- в обратном порядке, 486
- в прямом порядке, 486
- в ширину, 486
- сверху, 486
- симметричным способом, 486
- снизу, 486

Обход (вершин) дерева, 485
 Объединение, 170
 Объектный код, 36; 41
 файл, 279
 Объявление переменных, цикл, 97
 Оверлей, 277
 Оверлейная область памяти, 277
 Одинарная кавычка, специальная
 символьная константа, 62
 Одномерный массив, 108
 Односвязный список, 471
 Однострочные комментарии, 252
 Оператор
 #, 250
 ##, 250
 ., 75
 ?, 87
 ->, 75
 break, 102
 continue, 104
 defined, 249
 for, 92
 goto, 101
 if, 82
 return, 101; 157
 switch, 89
 выбора, 89
 доступа к члену структуры, 75; 172
 доступа через указатель, 75
 конкатенации, 250
 перехода, 101
 последовательного вычисления, 75
 превращения в строку, 250
 присваивания, 63
 склеивания, 250
 стрелка, 75
 точка, 75; 172
 Операции
 адресной арифметики, 130
 логические, 67
 поразрядные, 69
 сравнения, 67
 Операция
 &, 73
 *, 73
 ?, 72
 sizeof, 74
 определения размера, 74
 получения адреса, 73
 приведения типов, 78
 раскрытия ссылки, 73
 сравнение указателей, 130
 Орграф, 536

Ориентированный граф, 536
 Открытие файла, 219
 Отладка, 611
 Очередь, 460

П

Пакет переполнения, 505
 Параметр
 dwStyle, 586
 hMenu, 586
 hThisInst, 586
 hwnd, 588
 lpzClassName, 586
 nHow, 586
 nWinMode, 586
 Передача одномерного массива в
 функцию, 110
 Перекрестие, 585
 Перекрывающееся окно
 с обрамлением, 586
 Переменная епто, 408
 Переменное количество параметров,
 166
 Переменные списки аргументов, 263
 Переместимый код, 277
 Перенаправление ввода/вывода, 237
 Перенос программ, 610
 Переносимость, 33
 Перечисление, 170; 189
 Песочные часы, 585
 Пиктограмма по умолчанию, 584
 подача бумаги, специальная
 символьная константа, 62
 Поддерево, 484
 Поиск
 в глубину, 537
 в ширину, 546
 выбор метода, 556
 методом наискорейшего подъема, 548
 нескольких решений, 556
 с использованием частичного пути
 минимальной стоимости, 554
 Полный перебор, 546
 Порождающие правила, 514
 Последовательный поиск, 457
 Постфиксная запись, 468
 Постфиксный калькулятор, 469
 Поток, 217
 stderr, 237; 393
 stdin, 237

- stdout, 237
- Правила продвижения целых типов, 271
- Правило “неявного int”, 257
- Прагма
 - STDC CX_LIMITED_RANGE, 264
 - STDC FENV_ACCESS, 264
 - STDC FP_CONTRACT, 263
- Предварительный проход, 648
- Преобразование
 - типа указателя, 128
 - типов, 63; 77
- Префиксы типов, 590
- Присваивание
 - множественное, 64
 - составное, 65
 - указателей, 127
- Пробелы, 79
- Программный блок, 35
- Продвижение типов, 77
- Продукция, 514
- Проектирование сверху вниз, 592
- Пропуск лишних разделителей, 212
- Просмотр на одну лексему вперед, 647
- Прототип функции, 163
- Процедура окна, 579
- Прямой обход, 486
- Псевдокод, 593
- Пузырьковая сортировка, 440

Р

- Раздельная компиляция, 276
- Размерности массивов, 261
- Разреженная матрица, 494
- Разреженный массив, 494
- Расширенные целые типы, 270
- Расширенный тип
 - int_fast32_t, 270
 - int_least16_t, 270
 - int16_t, 270
 - intmax_t, 270
 - uintmax_t, 270
- Редактирование связей, 277
- Редактор связей, 41; 276
- Режим
 - a, 219
 - a+, 220
 - a+b, 220
 - ab, 220
 - g, 219
 - g+, 220
 - g+b, 220

- rb, 219
- w, 219
- w+, 220
- w+b, 220
- wb, 219
- Рекурсивное определение, 161
- Рекурсивный нисходящий синтаксический анализатор, 510
- Рекурсия, 161
- Рука, 585

С

- Связанный список, 471
- Сигнал, специальная символьная константа, 62
- Символ
 - информации, 584
 - ошибки, 584
- Символьные константы, 62
- Симметричный обход, 486
- Синтаксический
 - анализ методом рекурсивного спуска, 510
 - анализатор
 - выражений, 631
 - управляемый таблицей, 631
- Синтаксический разбор выражений, 510
- Системная программа, 36
- Системное меню, 586
- Соединение, 170
- Сортировка, 438
 - вставками, 444
 - количества сравнений, 445
 - преимущества, 445
 - дисковых файлов с произвольной выборкой, 454
 - критерии оценки алгоритма, 439
 - массивов, 439
 - методом пузырька, 440
 - посредством выбора, 443
 - пузырьком, 440
 - строк, 451
 - в лексикографическом порядке, 452
 - структур, 453
 - структур данных, 451
 - Шелла, 446
- Составной оператор, 105
- Составные литералы, 265
- Специальная символьная константа
 - \?, 62

- \\, 62
- \a, 62
- \b, 62
- \f, 62
- \n, 62
- \r, 62
- \t, 62
- \v, 62
- вертикальная табуляция, 62
- возврат каретки, 62
- восьмеричная, 62
- горизонтальная табуляция, 62
- двойная кавычка, 62
- знак вопроса, 62
- новая строка, 62
- обратный слэш, 62
- одинарная кавычка, 62
- подача бумаги, 62
- сигнал, 62
- удаление предыдущего символа, 62
- шестнадцатеричная, 62
- Спецификатор класса памяти
 - extern, 55
 - register, 59
 - static, 57
- Спецификатор преобразования
 - %%, 202, 209
 - %[], 209
 - %a, 202, 209
 - %c, 202, 209
 - %d, 202, 209
 - %e, 202, 209
 - %f, 202, 209
 - %g, 202, 209
 - %i, 202, 209
 - %n, 202, 209
 - %o, 202, 209
 - %p, 202, 209
 - %s, 202, 209
 - %u, 202, 209
 - %x, 202, 209
- Спецификатор типа
 - long, 45
 - short, 45
 - signed, 45
 - unsigned, 45
- Список параметров переменной
 - длины, 166
- Сравнение, 440
 - указателей, 130
- Старомодное объявление типа
 - функции, 165
- Стирание файлов, 227

- Строка, 108
- Строковые константы, 61
- Структура, 170
 - imaxdiv_t, 432
 - вложенная, 184
- Структура типа
 - div_t, 398
 - ldiv_t, 401
 - lldiv_t, 401

Т

- Текстовый поток, 217
- Тип
 - LPVOID, 586
 - LRESULT, 579
 - окна, 579
- Тип данных, 33
 - _Bool, 434
 - bool, 434
 - fenv_t, 430
 - fexcept_t, 430
 - intmax_t, 431
 - intptr_t, 432
 - jmp_buf, 402
 - mbstate_t, 418, 424
 - size_t, 418
 - uintmax_t, 431
 - uintptr_t, 432
 - va_list, 413
 - wchar_t, 418
 - wctrans_t, 418
 - wctype_t, 418
 - wint_t, 418
- Типы данных Windows, 580

У

- Удаление
 - вершин, 557
 - предыдущего символа, специальная
 - символьная константа, 62
 - путей, 557
- Указатель, 126
 - на массив, 109
 - на структуру, 181
 - на указатель, 134
 - на функцию, 137
 - операции над указателями, 129
 - преобразование типа, 128
 - текущей позиции, 217
 - файла, 219

Указатель-стрелка по умолчанию, 585
Универсальный решатель задач, 530
Упорядоченный обход, 486
Условная компиляция, 245
Условные операторы, 82

Ф

Файл, 217
Файлы описаний, 589
Факториал, 162; 532
Формальные параметры, 50
Форматный ввод/вывод, 201
Функции
 Windows API, 580
 ввода-вывода двухбайтовых
 символов, 420
 для преобразования формата
 целочисленных значений, 432
 для работы с массивами
 двухбайтовых символов, 423
 классификации символов широкого
 формата, 418
 обработки двухбайтовых символов,
 417
 преобразования строк двухбайтовых
 символов, 422
Функциональный блок, 620
Функция
 _exit(), 399
 _getch(), 198
 _getche(), 198
 a*b+c, 357
 a^b, 428
 abort, 392
 abs, 393
 acos, 347
 acosh, 348
 asctime, 374
 asin, 347; 348
 asinh, 349
 atan, 347; 349
 atan2, 347; 350
 atanh, 350
 atexit, 394
 atexit, 398; 399
 atof, 394
 atoi, 395
 atol, 396
 atoll, 396
 bsearch, 397
 btowc, 424

cabs, 426
cabsf, 426
cabsl, 426
cacos, 427
cacosf, 427
cacosh, 427
cacoshf, 427
cacoshl, 427
cacosl, 427
calloc, 386
carg, 427
cargf, 427
cargl, 427
casin, 427
casinf, 427
casinh, 427
casinhf, 427
casinhl, 427
casinl, 427
catan, 427
catanf, 427
catanh, 427
catanhf, 427
catanhl, 427
catanl, 427
cbrt, 351
ccos, 428
ccosf, 428
ccosh, 428
ccoshf, 428
ccoshl, 428
ccosl, 428
ceil, 347; 351
cexp, 428
cexpf, 428
cexpl, 428
cimag, 428
cimagf, 428
cimagl, 428
clock, 375
clog, 428
clogf, 428
clogl, 428
conj, 428
conjf, 428
conjl, 428
copysign, 352
cos, 347; 352
cosh, 347; 353
cpow, 428
cpowf, 428
cpowl, 428
cproj, 429

cprojf, 429
 cprojl, 429
 creal, 429
 crealf, 429
 creall, 429
 CreateWindow(), 586
 csin, 429
 csinf, 429
 csinh, 429
 csinhf, 429
 csinhl, 429
 csinl, 429
 csqrt, 429
 csqrtf, 429
 csqrtl, 429
 ctan, 429
 ctanf, 429
 ctanh, 429
 ctanhf, 429
 ctanhl, 429
 ctanl, 429
 ctime, 375
 difftime, 376
 div(), 398
 e^{arg} , 428
 $e^{arg}-1$, 355
 erf, 353
 erfc, 354
 exit, 103; 398
 exp, 347; 354
 exp2, 355
 expm1, 355
 fabs, 347; 355
 fclose, 218; 220; 285
 fdim, 356
 feclearexcept, 430
 fegetenv, 430
 fegetexceptflag, 430
 fegetround, 430
 feholdexcept, 431
 feof, 218; 223; 286
 feraiseexcept, 430
 ferror, 218; 226; 287
 fesetenv, 431
 fesetexceptflag, 430
 fesetround, 430
 fetestexcept, 430
 feupdateenv, 431
 fflush, 218; 287
 fgetc, 218; 221; 288; 420
 fgetpos, 289
 fgets, 199; 218; 224; 289
 fgetwc, 420

floor, 347; 356
 fma, 357
 fmax, 357
 fmin, 357
 fmod, 347; 358
 fopen, 218; 219; 290
 fprintf, 218; 235; 291; 420
 fputc, 218; 221; 292; 420
 fputs, 218; 224; 293; 420
 fputwc, 420
 fputws, 420
 fread, 228; 293
 free, 140; 387
 freopen, 238; 294
 frexp, 347; 358
 fscanf, 218; 235; 295; 420
 fseek, 218; 234; 296
 fsetpos, 297
 ftell, 218; 298
 fwide, 421
 fwprintf, 420
 fwrite, 228; 298
 fwscanf, 420
 getc, 218; 221; 299; 420
 getchar, 197; 300; 420
 getenv, 399
 gets, 199; 300
 GetStockObject(), 585
 getwc, 420
 getwchar, 420
 gmtime, 377
 hypot, 359
 ilogb, 359
 imaxabs, 432
 imaxdiv, 432
 isalnum, 320; 418
 isalpha, 321; 418
 isblank, 322; 418
 iscntrl, 418
 iscntrl, 322
 isdigit, 323; 418
 isgraph, 324; 419
 islower, 324; 419
 isprint, 325; 419
 ispunct, 325; 419
 isspace, 326; 419
 isupper, 327; 419
 iswalnum, 418
 iswalpha, 418
 iswblank, 418
 iswcntrl, 418
 iswctype, 419
 iswdigit, 418

strtol, 410; 423; 433
 strtold, 411; 423
 strtoll, 411; 423
 strtoul, 411; 423; 433
 strtoull, 412; 423
 strtoumax, 433
 strxfrm, 342; 422
 swprintf(), 420
 system(), 413
 tan, 347; 370
 tanh, 347; 371
 tgamma, 371
 time, 384
 tmpfile, 315
 tmpnam, 315
 tolower, 342; 419
 toupper, 343; 419
 towctrans(), 419
 tolower, 419
 towupper, 419
 trunc, 372
 ungetc, 316; 420
 ungetwc, 420
 UpdateWindow(), 587
 val * FLT_RADIX^{exp}, 368
 vfprintf, 317; 421
 vfscanf, 318; 421
 vfwprintf, 421
 vfwscanf, 421
 vprintf, 317; 421
 vscanf, 318; 421
 vsnprintf, 317
 vsprintf, 317; 421
 vsscanf, 318; 421
 vswprintf, 421
 vswscanf, 421
 vwprintf, 421
 vwscanf, 421
 wctomb, 424
 wscat, 422
 wcschr, 422
 wcscmp, 422
 wscoll, 422
 wcsncpy, 422
 wcsncpy, 422
 wcsftime, 423
 wcslen, 422
 wcsncat, 422
 wcsncmp, 422
 wcsncpy, 422
 wcsprbrk, 422
 wcsrchr, 422
 wcsrtombs, 424
 wcsspncpy, 422
 wcsstr, 422
 wcstod, 423
 wcstof, 423
 wcstol, 423
 wcstold, 423
 wcstoll, 423
 wcstombs, 424
 wcstombs, 415
 wcstoul, 423
 wcstoul, 423
 wcstoull, 423
 wcstoumax, 433
 wcsxfrm, 422
 wctob, 424
 wctomb, 424
 wctomb, 415
 wctrans, 419
 wctype, 419
 wctype, 419
 WinMain, 579
 wmemchr, 423
 wmemcmp, 423
 wmemcpy, 423
 wmemmove, 423
 wmemset, 423
 wprintf, 421
 wscanf, 421
 абсолютная величина, 355; 426
 абсолютная величина значение, 432
 абсолютное значение
 целочисленного аргумента, 393
 аргумент комплексного числа, 427
 аргументы, 149
 арккосинус, 348; 427
 арксинус, 349; 427
 арктангенс, 349; 427
 отношения, 350
 вещественная часть, 429
 возврат
 в вызывающую программу, 157
 возврат значений, 158
 указателей, 160
 времени и даты, 374
 вызов
 по значению, 149
 по ссылке, 150
 с помощью массива, 152
 гамма-функция, 371

гиперболический
 арккосинус, 348; 427
 арксинус, 349; 427
 арктангенс, 350; 427
 косинус, 353; 428
 синус, 369; 429
 тангенс, 371; 429
Г-функция, 360
Г-функция Эйлера, 360
двоичный логарифм, 363
десятичный логарифм, 362
динамического выделения памяти, 140
длина гипотенузы, 359
дополнительный интеграл
вероятности, 354
дробная часть, 364
интеграл
 вероятности, 354
 Гаусса, 354
 ошибок, 354
квадратный корень, 370; 429
 из суммы квадратов, 359
комплексное сопряжение, 428
косинус, 352; 428
кубический корень, 351
логарифм по основанию 10, 362
логарифм по основанию 2, 363
максимум, 357
мантисса, 358
минимум, 357
мнимая часть, 428
модуль, 426
натуральный логарифм, 361; 362; 428
натуральный логарифм гамма-функции, 360
нормального распределения, 354
область действия, 148
обратного вызова, 579
общий вид, 148
окна, 579
округление до ближайшего целого, 360; 361; 363; 364; 365; 367
остаток, 398; 401; 432
остаток от деления, 358; 366; 367
отбрасывание дробной части, 372
ошибок, 354
показатель, 363
порядок, 359
проекция сферу Римана, 429
прототип, 163

рекурсивная, 161
синус, 368; 429
сравнения, 397; 404
старомодное объявление, 165
степень, 366
 двойки, 355
тангенс, 370; 429
типа void, 161
целая часть, 364
целый показатель степени числа 2, 358
частное, 367; 398; 401; 432
эйлеров интеграл второго рода, 360
экспонента, 354

Х

Хоар, 448
Хэширование, 503

Ц

Целочисленное расширение, 77
Цепочка хэширования, 505
Цикл
 do-while, 100
 for, 97
 while, 98
 обработки сообщений, 587
 объявление переменных, 97
Циклическая очередь, 464

Ч

Числовые выражения, 510
Чтение символа, 221

Ш

Шейкер-сортировка, 443
 время выполнения, 443
Шелл Дональд Л., 446
шестнадцатеричная константа,
специальная символьная константа, 62
Шестнадцатеричные константы, 61

Э

Эвристика, 531
Элемент управления, 576

Научно-популярное издание

Герберт Шилдт

Полный справочник по С, 4-е издание

Литературный редактор	<i>С.Г. Татаренко</i>
Верстка	<i>О.В. Линник</i>
Художественный редактор	<i>В.Г. Павлютин</i>
Технический редактор	<i>Г.Н. Горобец</i>
Корректор	<i>Л.А. Гордиенко, Т.А. Корзун, О.В. Мишутина</i>

Издательский дом “Вильямс”.
101509, Москва, ул. Лесная, д. 43, стр. 704.
Изд. лиц. ЛР № 090230 от 23.06.99
Госкомитета РФ по печати.

Подписано в печать 17.12.2001. Формат 70×100/16.
Гарнитура Times. Печать офсетная.
Усл. печ. л. 56,7. Уч.-изд. л. 33.
Тираж 4000 экз. Заказ № 2430.

Налоговая льгота – общероссийский классификатор продукции ОК 005-93,
том 2: 953000 – книги, брошюры.

Отпечатано с диапозитивов в ФГУП “Печатный двор”
Министерства РФ по делам печати,
телерадиовещания и средств массовых коммуникаций.
197110, Санкт-Петербург, Чкаловский пр., 15.



Брайан Хетч, Джеймс Ли,
Джордж Курц

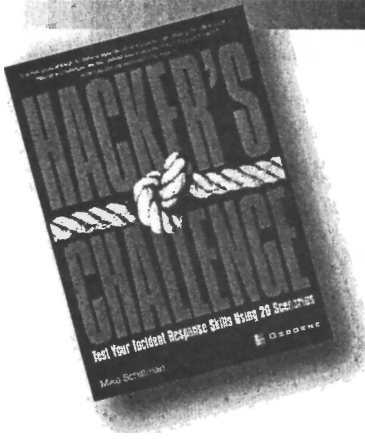
Секреты хакеров. Безопасность Linux — готовые решения

В продаже

Современная вычислительная техника и компьютерные сети подвергаются различным угрозам со стороны нечистоплотных пользователей. Особенно много проблем вызывает механизм защиты операционной системы Linux. Сегодня незащищенные версии Linux представляют собой одну из наиболее уязвимых для атаки целей во всем киберпространстве. Данную книгу можно назвать продолжением всемирно известного бестселлера Секреты хакеров. Безопасность сетей — готовые решения, 2-е издание, в которой все внимание сосредоточено на безопасности при работе в ОС Linux. Ее авторы уже много лет считаются ведущими и признанными специалистами по защите компьютерных систем. Это позволило им рассмотреть проблемы хакинга в Linux на новом, не имеющем аналогов уровне. Книга относится к тому редкому типу книг, которые наглядно объясняют, что именно происходит, когда злоумышленники атакуют системы Linux. Читателям продемонстрировано, чем Linux отличается от других Unix-подобных систем, раскрыты хакерские методы всех типов атак, которые используются для несанкционированного доступа к системам Linux, нарушения работы их служб и взлома компьютерных сетей. Детально изучены средства противодействия атакам хакеров и методы оперативного выявления вторжения. В этой книге нет пустых мест — после описания реальных листингов выполняемых атак предоставляются такие же реальные рецепты отражения каждой конкретной атаки.

Материал книги изложен простым и доступным языком, с использованием наглядных примеров, поэтому книга будет полезна самому широкому кругу читателей — начиная от обычных пользователей домашних компьютеров и заканчивая высококвалифицированными системными администраторами крупных компаний.

Майк Шиффман



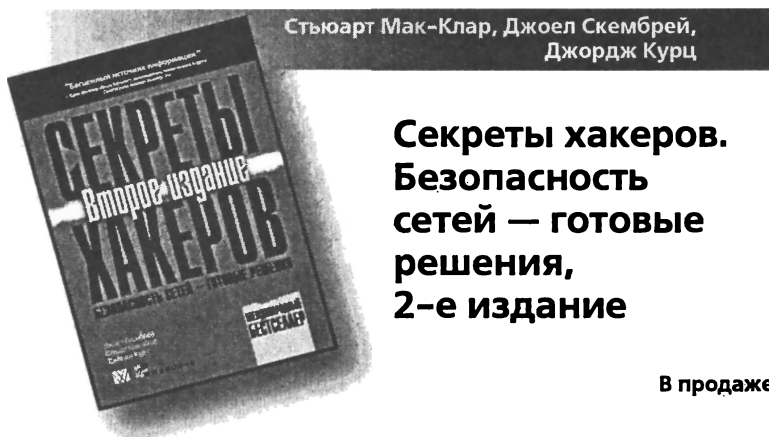
Защита от хакеров: 20 сценариев взлома

Плановая дата выхода
2-й кв. 2002 г.

Что приводит к инциденту? Из-за чего он происходит? Что способствует ему? Как его можно избежать? Каким образом уменьшить ущерб? И, самое главное, *как это случилось?* Если вас интересуют ответы на такие вопросы, то это книга для вас.

Здесь вы найдете истории взломов, основанные на фактах и изложенные ведущими исследователями, консультантами и судебными аналитиками, работающими в современной индустрии компьютерной безопасности. Эта книга не похожа на остальные современные книги, посвященные хакерам. Но в книге не просто пересказываются случаи взлома — здесь предоставлена их подноготная. В ходе изложения каждой истории читатель ознакомится с информацией об инциденте и узнает способы его предотвращения.

Книга состоит из двух частей. В первой части приводится описание случая взлома, а также все необходимые сведения (системные журналы и т.д.), необходимые читателю для создания полной картины инцидента. Затем формулируются специфические вопросы, с помощью которых можно более детально проанализировать описанный инцидент. Во второй части каждый случай рассматривается достаточно подробно и даются ответы на поставленные вопросы.



Стьюарт Мак-Клар, Джоел Скембрей,
Джордж Курц

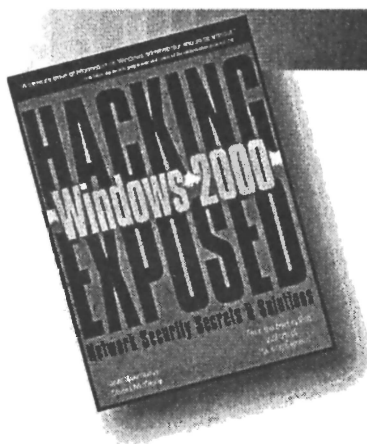
Секреты хакеров. Безопасность сетей — готовые решения, 2-е издание

В продаже

Эта книга появилась из-за того, что общедоступная информация вовсе не обязательно автоматически попадает в руки большинства людей. Секреты хакеров — это очень подробная информация о способах обеспечения компьютерной безопасности. В ней приведено полное описание изъянов в системах защиты: что они собой представляют, как их можно использовать и какие контрмеры необходимо предпринимать. После чтения этой книги вы будете знать о своей сети гораздо больше и, что еще важнее, сможете защитить ее гораздо лучше, чем с помощью сведений из любых других аналогичных изданий. Эта книга содержит по-настоящему бесценную информацию. Здесь приведены сведения для тех, кому необходимо знать о том, какие действия предпринимают хакеры, как функционируют используемые при этом средства и какие изъяны скрыты в системе защиты используемых сетей.

Первое издание этой книги стало настоящим бестселлером среди книг на компьютерную тематику: свыше 70 тысяч экземпляров продано менее чем за год. Авторам пришлось очень быстро обновить содержимое книги, а это говорит о том, что появилось так много новой информации, что понадобилось второе издание. В книге описаны общие принципы атак хакеров и способы защиты от них. Она может быть полезна администраторам, отвечающим за защиту сетей, программистам, стремящимся обеспечить безопасность своих программ, а также всем, кто интересуется вопросами безопасности сетей. В книге не только подробно описаны возможные бреши в защите сетей, но и даны исчерпывающие рекомендации по их обнаружению и устранению. Особо следует подчеркнуть системный подход к анализу сетевого хакинга, включающий все этапы подготовки и реализации атак, начиная с предварительного сбора данных и заканчивая проникновением в систему и получением ценной информации. Во второе издание книги включено много нового материала (в том числе по Windows 2000), а также описаны новейшие средства и программы взлома, поэтому его можно порекомендовать даже тем читателям, которые знакомы с первым изданием.

Книга рассчитана на широкий круг читателей — от новичков до профессионалов, работающих с различными операционными системами.



Джоел Скембрей,
Стьюарт Мак-Клар

Секреты хакеров: безопасность Windows 2000 – готовые решения

Плановая дата выхода
2-й кв. 2002 г.

Особенностям защиты операционной системы Windows 2000 в известном бестселлере *Секреты хакеров. Безопасность сетей – готовые решения. 2-е издание*, была посвящена отдельная глава, однако вполне очевидно, что эта тема заслуживает более глубокого рассмотрения.

Данную книгу можно назвать специализированным продолжением упомянутого выше всемирно известного бестселлера, где все внимание сосредоточено на безопасности работы в ОС Windows 2000. Узкая специализация позволила авторам глубже проанализировать механизмы защиты этой операционной системы и предоставить читателям большой объем полезной информации. Даже специалисты корпорации Microsoft найдут в этой книге новые полезные сведения об особенностях защиты Windows-систем.

Не вызывает сомнения, что эта книга будет полезнейшим инструментом в арсенале средств каждого системного администратора, ответственного за безопасность вверенной ему информационной инфраструктуры.

Четвертое издание

Полный справочник по С

Содержит описание C99 — нового стандарта языка С, одобренного комитетами ANSI/ISO

Содержит описание C99 — нового стандарта языка С, одобренного комитетами ANSI/ISO

Наиболее полный ресурс по С — пересмотренное и дополненное издание

Герберт Шилдт, ведущий автор книг по программированию мирового уровня, переработал свой не устаревающий справочник-бестселлер по С и включил в него самую свежую информацию по C99 — новому стандарту языка С, принятому комитетами ANSI и ISO. И умудренный опытом профессионал в области языка С, и новичок найдет в нем полные ответы на все свои вопросы, касающиеся языка С. В этом авторитетном руководстве Шилдт подробно описывает все особенности языка С, его библиотеки, приложения, дает профессиональные советы, приводит сотни примеров с объяснениями эксперта. А в заключительной

главе автор преподносит особо ценный подарок — он приглашает читателя принять участие в разработке увлекательного проекта — в создании интерпретатора языка С, который читатель затем сможет использовать в готовом виде или дополнить его своими функциями. Все объяснения даются в стиле Шилдта — доходчиво, кратко, точно, — именно в том стиле, за который Герберта так любят миллионы читателей!



В книге:

- полное описание языка С, включая как его первоначальный стандарт C89, так и новые средства, добавленные в C99
- подробные объяснения всех ключевых слов, типов данных и операторов
- описание указателей, средств дискового ввода-вывода и динамического распределения памяти на профессиональном уровне
- подробная информация о всех функциях библиотеки С
- знакомство с новыми средствами, добавленными в C99; среди них — применение restrict к указателям, зарезервированное слово inline, массивы переменной длины и типы данных long long
- реально используемые алгоритмы, структуры данных и приложения; среди них — стеки, очереди, деревья, разреженные массивы и алгоритмы сортировки; приведены также все необходимые сведения об алгоритмах поиска, использующих методы искусственного интеллекта
- советы по эффективному использованию среды программирования на С
- советы по переносу программ и их отладке
- полный исходный код интерпретатора языка С, который можно использовать без каких-либо изменений или дополнить его функциями, необходимыми для решения конкретных задач

Герберт Шилдт — ведущий специалист в области программирования на языке С; член комитетов ANSI/ISO, которыми разрабатывался и принимался стандарт языка С. Шилдт — автор таких книг, как *Teach Yourself C*, *C++: The Complete Reference*, *Windows 2000 Programming from the Ground Up*, *Полный справочник по Java* и многих других бестселлеров.

ПО С

Четвертое издание

Категория: программирование на языке С
Уровень: от начинающих до опытных программистов



Издательский дом "Вильямс"

<http://www.willamspublishing.com>

OSBORNE

A Division of The McGraw-Hill Companies



ISBN 5-8459-0226-6



9 785845 902269



OSBORNE

