Харви Дейтел, Пол Дейтел

КАК ПРОГРАММИРОВАТЬ НА С

Книга предлагает читателю курс программирования, ориентированный на языки С/С++, и рассчитана как на начинающих, не владеющих никакими языками программирования, так и на опытных программистов, которые могут просто пропустить не интересующие их главы. Помимо достаточно полного и глубокого изложения языка С дается весьма серьезное введение в С++, одного из наиболее перспективных на настоящий момент языков; ему посвящена значительная часть книги. Особое внимание уделяется методикам структурного и объектно-ориентированного программирования больших программных систем. Примеры и многочисленные упражнения знакомят читателя с часто применяемыми алгоритмами и фундаментальными структурами данных, показывая технические приемы их реализации. Приводится также масса полезных советов.

Книга адресована широкому кругу читателей, от новичков до студентов, изучающих программирование в рамках своей специальности.

Содержание

Предисловие 17
Об этой книге 18
Обзор книги 20
Глава 1. Принципы машинной обработки данных 29
1.1. Введение 30
1.2. Что такое компьютер? 32
1.3. Внутренняя организация компьютера 33
1.4. Пакетная обработка, мультипрограммирование и разделение времени 34
1.5. Персональные вычисления, распределенные вычисления и вычисления в модели клиент/сервер 34
1.6. Машинные языки, языки ассемблера и языки высокого уровня 35
1.7. История языка С 36
1.8. Стандартная библиотека С 37
1.9. Другие языки высокого уровня 38
1.10. Структурное программирование 38
1.11. Основные принципы среды С 39
1.12. Общие замечания о С и этой книге 41
1.13. Concurrent C (Параллельный С) 42
1.14. Объектно-ориентированное программирование и С++ 43
Резюме Хороший стиль программирования Советы по переносимости программ Советы по повышению эффективности Упражнения для самоконтроля Ответы к упражнениям для самоконтроля Упражнения Рекомендуемая литература
Глава 2. Введение в программирование на С 53
2.1. Введение 54
2.2. Простая программа на С: печать строки текста 54

2.3. Еще одна простая программа на С; сложение двух целых чисел 58
2.4. Общие понятия о памяти компьютера 63
2.5. Арифметика в С 64
2.6. Принятие решений: операции равенства и отношения 67
Резюме Распространенные ошибки программирования Хороший стиль программирования Советы по переносимости программ Упражнения для самоконтроля Ответы на упражнения для самоконтроля Упражнения
Глава 3. Структурная разработка программ 87
3.1. Введение 88
3.2. Алгоритмы 88
3.3. Псевдокод 89
3.4. Управляющие структуры 90
3.5. Структура выбора if 92
3.6. Структура выбора if/else 94
3.7. Структура повторения while 98
3.8. Формулирование алгоритмов: пример 1 (повторение, управляемое счетчиком) 99
3.9. Формулирование алгоритмов на основе нисходящего пошагового уточнения: пример 2 (повторение, управляемое контрольным значением) 102
3.10. Формулирование алгоритмов на основе нисходящего пошагового уточнения: пример 3 (вложенные управляющие структуры) 108
3.11. Операции присваивания 113
3.12. Операции инкремента и декремента 114
Резюме Распространенные ошибки программирования Хороший стиль программирования Советы, по повышению эффективности Общие методические замечания Упражнения для самоконтроля Ответы к упражнениям для самоконтроля Упражнения
Глава 4. Управление программой 137
4.1. Введение 138
4.2. Основы структур повторения 139
4.3. Повторение, управляемое счетчиком 139
4.4. Структура повторения for 141
4.5. Структура for: замечания и рекомендации 144
4.6. Примеры структур for 145
4.7. Структура со множественным выбором switch 148
4.8. Структура повторения do/while 154
4.9. Операторы break и continue 156
4.10. Логические операции 158
4.11. Смешивание операций равенства (==) и присваивания (=) 161
4.12. Краткая сводка по структурному программированию 163
Резюме Распространенные ошибки программирования Хороший стиль

программирования Советы по повышению эффективности Советы по переносимости программ Общие методические замечания Упражнения для самоконтроля Ответы к упражнениям для самоконтроля Упражнения
Глава 5. Функции 185
5.1. Введение 186
5.2. Программные модули в C 187
5.3. Функции математической библиотеки 188
5.4. Функции 189
5.5. Определения функций 190
5.6. Прототипы функций 194
5.7. Заголовочные файлы 197
5.8. Вызов функций: вызов по значению и по ссылке 198
5.9. Генерация случайных чисел 199
5.10. Пример: стохастическая игра 203
5.11. Классы памяти 206
5.12. Правила области действия 209
5.13. Рекурсия 210
5.14. Пример применения рекурсии: числа Фибоначчи 215
5.15. Рекурсия в сравнении с итерацией 219
Резюме Распространенные ошибки программирования Хороший стиль программирования Советы по переносимости программ Советы по повышению эффективности Общие методические замечания Упражнения для самоконтроля Ответы на упражнения для самоконтроля Упражнения
Глава 6. Массивы 247
6.1. Введение 248
6.2. Массивы 249
6.3. Объявление массивов 251
6.4. Примеры работы с массивами 251
6.5. Передача массивов в функции 263
6.6. Сортировка массивов 267
6.7. Пример: вычисление среднего значения, медианы и наиболее вероятного значения с использованием массивов 269
6.8. Поиск в массивах 273
6.9. Многомерные массивы 277
Резюме Распространенные ошибки программирования Хороший стиль программирования Советы по повышению эффективности Общие методические замечания Упражнения для самоконтроля Ответы к упражнениям для самоконтроля Упражнения Упражнения на рекурсию
Глава 7. Указатели 307
7.1. Введение 308
7.2. Объявление и инициализация переменной-указателя 309

7.3. Операции над указателями 310
7.4. Передача параметра по ссылке 312
7.5. Использование модификатора const с указателями 316
7.6. Программа пузырьковой сортировки, использующая вызов по ссылке 322
7.7. Выражения и арифметические операции с указателями 326
7.8. Связь между указателями и массивами 329
7.9. Массивы указателей 333
7.10. Пример: программа тасовки и сдачи колоды карт 334
7.11. Указатели на функции 339
Резюме Распространенные ошибки программирования Хороший стиль программирования Советы по повышению эффективности Советы по переносимости программ Общие методические замечания Упражнения для самоконтроля Ответы на упражнения для самоконтроля Упражнения Специальный раздел: как самому построить компьютер
Глава 8. Символы и строки 369
8.1. Введение 370
8.2. Строки и символы 370
8.3. Библиотека обработки символов 372
8.4. Функции преобразования строк 377
8,5. Функции стандартной библиотеки ввода/вывода 382
8,6. Функции операций над строками из библиотеки обработки строк 385
8.7. Функции сравнения из библиотеки обработки строк 388
8.8. Функции поиска из библиотеки обработки строк 390
8.9. Функции памяти библиотеки обработки строк 395
8.10. Другие функции из библиотеки обработки строк 399
Резюме Распространенные ошибки программирования Хороший стиль программирования Советы по переносимости программ Упражнения для самоконтроля Ответы к упражнениям для самоконтроля Упражнения Специальный раздел: более сложные упражнения по работе со строками
Глава 9. Форматированный ввод/вывод 419
9.1. Введение 420
9.2. Потоки 421
9.3. Форматированный вывод с применением printf 421
9.4. Печать целых чисел 422
9.5. Печать чисел с плавающей точкой 423
9.6. Печать строк и символов 425
9.7. Другие спецификаторы преобразования 426
9.8. Печать с заданием ширины поля и точности представления 428
9.9. Использование флагов в строке управления форматом printf 430
9.10. Печать литералов и Esc-последовательностей 433
9.11. Форматированный ввод с применением scanf 434
Резюме Распространенные ошибки программирования Хороший стиль

программирования Советы по переносимости программ Упражнения для самоконтроля Ответы на упражнения для самоконтроля Упражнения
Глава 10. Структуры, объединения, операции с битами и перечисления. 451
10.1. Введение 452
10.2. Описания структур 453
10.3. Инициализация структур 455
10.4. Доступ к элементам структур 455
10.5. Использование структур с функциями 457
10.6. Typedef 458
10.7. Пример: моделирование высокоэффективной тасовки и раздачи карт 459
10.8. Объединения 461
10.9. Поразрядные операции 463
10.10. Битовые поля 471
10.11. Перечислимые константы 474
Резюме Распространенные ошибки программирования Хороший стиль программирования Советы по переносимости программ Советы по повышению эффективности Общие методические замечания Упражнения для самоконтроля Ответы на упражнения для самоконтроля Упражнения
Глава 11. Работа с файлами 489
11.1. Введение 490
11.2. Иерархия данных 490
11.3. Файлы и потоки 493
11.4. Создание файла последовательного доступа 493
11.5. Чтение данных из файла последовательного доступа 499
11.6. Файлы произвольного доступа 503
11.7. Создание файла произвольного доступа 504
11.8. Произвольная запись данных в файл произвольного доступа 506
11.9. Последовательное чтение данных из файла произвольного доступа 508
11.10. Пример: программа обработки транзакций 508
Резюме Распространенные ошибки программирования Хороший стиль программирования Советы, по переносимости программ Советы по повышению эффективности Упражнения для самоконтроля Ответы на упражнения для самоконтроля Упражнения
Глава 12. Структуры данных 527
12.1. Введение 528
12.2. Структуры, ссылающиеся на себя 529
12.3. Динамическое распределение памяти 530
12.4. Связанные списки 531
12.5. Стеки 539
12.6. Очереди 545
12.7. Деревья 550

Резюме Распространенные ошибки программирования Хороший стиль программирования Советы по повышению эффективности Советы по переносимости программ Упражнения для самоконтроля Ответы на упражнения для самоконтроля Упражнения
Глава 13. Препроцессор 587
13.1. Введение 588
13.2. Директива препроцессора #include 588
13.3. Директива препроцессора #define: символические константы 589
13.4. Директива препроцессора #define: макросы 590
13.5. Условная компиляция 592
13.6. Директивы препроцессора #error и #pragma 593
13.7. Операции # и ## 593
13.8. Нумерация строк 594
13.9. Предопределенные символические константы 594
13.10. Макрос подтверждения 594
Резюме Распространенные ошибки программирования Хороший стиль программирования Советы по повышению эффективности Упражнения для самоконтроля Ответы на упражнения для самоконтроля Упражнения
Глава 14. Специальные вопросы 601
14.1. Введение 602
14.2. Переадресация ввода/вывода в системах UNIX и DOS 602
14.3. Списки аргументов переменной длины 603
14.4. Аргументы командной строки 606
14.5. Замечания относительно компиляции программ из нескольких исходных файлов 607
14.6. Выход из программы с помощью exit и atexit 609
14.7. Модификатор типа volatile 609
14.8. Суффиксы для целых констант и констант с плавающей точкой 610
14.9. Еще раз о файлах 611
14.10. Обработка сигналов 613
14.11. Динамическое выделение памяти: функции calloc и realloc 615
14.12. Безусловный переход: goto 616
Резюме Распространенные ошибки программирования Советы по переносимости программ Советы по повышению эффективности Общие методические замечания Упражнения для самоконтроля Ответы, на упражнения для самоконтроля Упражнения
Глава 15. C++ как “улучшенный” C 625
15.1. Введение 626
15.2. Однострочные комментарии C++ 627
15.3. Потоковый ввод/вывод C++ 628
15.4. Объявления в C++ 630
15.5. Создание новых типов данных в C++ 630

15.6. Прототипы функций и контроль соответствия типов 631
15.7. Встроенные функции 633
15.8. Параметры-ссылки 637
15.9. Модификатор const 641
15.10. Динамическое распределение памяти с помощью new и delete 643
15.11. Параметры, используемые по умолчанию 644
15.12. Унарная операция разрешения области действия 646
15.13. Перегрузка функций 647
15.14. Спецификации внешней связи 649
15.15. Шаблоны функций 649
Резюме Распространенные ошибки программирования Хороший стиль программирования Советы по переносимости программ Советы по повышению эффективности Общие методические замечания Упражнения для самоконтроля Ответы к упражнениям для самоконтроля Упражнения Рекомендуемая литература Приложение: ресурсы C++
Глава16. Классы и абстракция данных. 663
16.1. Введение 664
16.2. Определение структур 666
16.3. Доступ к элементам структур 667
16.4. Реализация пользовательского типа Time с помощью структуры 668
16.5. Реализация абстрактного типа данных Time с помощью класса 670
16.6. Область действия класса и доступ к элементам класса 676
16.7. Отделение интерфейса от реализации 677
16.8. Управление доступом к элементам класса 681
16.9. Функции доступа и сервисные функции 684
16.10. Инициализация объектов класса: конструкторы 687
16.11. Использование с конструкторами аргументов по умолчанию 688
16.12. Деструкторы 691
16.13. Когда вызываются конструкторы и деструкторы 691
16.14. Использование элементов данных и элементов-функций 694
16.15. Скрытая ловушка: возвращение ссылки на закрытый элемент данных 699
16.16. Присваивание по умолчанию путем поэлементного копирования 701
16.17. Повторное использование программного обеспечения 701
Резюме Распространенные ошибки программирования Хороший стиль программирования Советы, по повышению эффективности Общие методические замечания Упражнения для самоконтроля Ответы, к упражнениям для самоконтроля Упражнения
Глава 17. Классы: часть II 715
17.1. Введение 716
17.2. Константные объекты и константные элементы-функции 717
17.3. Композиция: классы в качестве элементов других классов 723
17.4. Дружественные функции и дружественные классы 727

17.5. Указатель this 729
17.6. Динамическое распределение памяти с помощью операций new и delete 734
17.7. Статические элементы класса 735
17.8. Абстракция данных и сокрытие информации 739
17.9. Контейнерные классы и итераторы 743
17.10. Шаблоны классов 743
Резюме Распространенные ошибки программирования Хороший стиль программирования Советы по повышению эффективности Советы, по переносимости программ Общие методические замечания Упражнения для самоконтроля Ответы к упражнениям для самоконтроля Упражнения
Глава 18. Перегрузка операций 755
18.1. Введение 756
18.2. Основные принципы перегрузки операций 757
18.3. Запреты на перегрузку операций 759
18.4. Функции-операции как элементы класса и как дружественные функции 760
18.5. Перегрузка операций передачи в поток и извлечения из потока 762
18.6. Перегрузка одноместных операций 764
18.7. Перегрузка двухместных операций 765
18.8. Пример: класс Array 765
18.9. Преобразование типов 776
18.10. Пример: класс String 777
18.11. Перегрузка ++ и -- 786
18.12. Пример: класс Date 788
Резюме Распространенные ошибки программирования Хороший стиль программирования Советы по повышению эффективности Общие методические замечания Упражнения для самоконтроля Ответы к упражнениям для самоконтроля Упражнения
Глава 19. Наследование 805
19.1. Введение 806
19.2. Базовые и производные классы 808
19.3. Защищенные элементы 810
19.4. Приведение указателей базового класса к указателям на производный класс 811
19.5. Применение функций-элементов 815
19.6. Переопределение элементов базового класса в производном классе 816
19.7. Открытые, защищенные и закрытые базовые классы 820
19.8. Непосредственные и косвенные базовые классы 820
19.9. Применение конструкторов и деструкторов в производном классе 821
19.10. Неявное преобразование объектов производного класса к базовому 824
19.11. Наследование в конструировании программного обеспечения 825
19.12. Композиция в сравнении с наследованием 827
19.13. Отношения “использует” и “знает” 828

19.14. Пример: Point, Circle, Cylinder 828
19.15. Сложное наследование 833
Резюме Распространенные ошибки программирования Хороший стиль программирования Совет по повышению эффективности Общие методические замечания Упражнения для самоконтроля Ответы на упражнения для самоконтроля Упражнения
Глава 20. Виртуальные функции и полиморфизм 847
20.1. Введение 848
20.2. Обработка различных типов данных при помощи операторов switch 849
20.3. Виртуальные функции 849
20.4. Абстрактные базовые классы и конкретные классы 850
20.5. Полиморфизм 851
20.6. Пример: программа начисления заработной платы, использующая возможности полиморфизма 853
20.7. Новые классы и динамическое связывание 862
20.8. Виртуальные деструкторы 864
20.9. Пример: наследование интерфейса и реализации 864
Резюме Распространенные ошибки программирования Хороший стиль программирования Советы по повышению эффективности Общие, методические замечания Упражнения для самоконтроля. Ответы на упражнения для самоконтроля Упражнения
Глава 21. Потоки ввода/вывода в C++ 877
21.1. Введение 879
21.2. Потоки 879
21.3. Потоковый вывод 882
21.4. Потоковый ввод 886
21.5. Функции неформатируемого ввода/вывода read, gcount и write 892
21.6. Манипуляторы потоков 893
21.7. Флаги форматирования потока 896
21.8. Состояния ошибки потоков 906
21.9. Ввод/вывод определяемых пользователем типов 907
21.10. Привязка потока вывода к потоку ввода 910
Резюме Распространенные ошибки программирования Хороший стиль программирования Советы, по повышению эффективности Общие методические замечания Упражнения для самоконтроля Ответы на упражнения для самоконтроля Упражнения
Приложение А. Синтаксис C 929
Приложение Б. Стандартная библиотека 941
Приложение В. Таблица приоритета операций 978
Приложение Г. Набор символов ASCII 980
Приложение Д. Системы счисления 981
Д.1. Введение 982
Д.2. Запись двоичных чисел в виде восьмеричных и шестнадцатеричных 985

Д.3. Преобразование восьмеричных и шестнадцатеричных чисел в двоичные 987
Д.4. Преобразование числа из двоичной, восьмеричной или шестнадцатеричной форм в десятичную 987
Д.5. Преобразование десятичного числа в двоичное, восьмеричное или шестнадцатеричное 988
Д.6. Отрицательные двоичные числа: дополнение до двух 990
Резюме Упражнения для самоконтроля Ответы на упражнения для самоконтроля Упражнения

Предметный указатель

#define 589, 590
#define NDEBUG 594
#elif 592
#else 592
#endif 592
#error 593
#if 592
#ifdef 592
#ifndef 592
#ftinclude "filename" 588
#include <filename> 588
#line 594
#pragma 593
#undef 591
в качестве точности 430
в качестве ширины поля 430
\ (обратная дробная черта, символ продолжения) 591
DATE 595
FILE 595
LINE 595
STDC 595
TIME 595
<ctrl-z> 151
<return><ctrl-d> 151
A—B
a.out в UNIX 589
a[i] 249
a[i][j] 277
abort 594
argc 606
argv 606
ASCII 389
asm 636
assert 594
assert.h 594
atexit 609
atof 378
atoi 378
atol 379
badbit 886, 906
break 151, 156
C—D
C 36
C++ 43, 626
calloc 615
catch 636
cerr 880, 882
char 154, 196
cin (входной поток) 628, 880, 881
class 630, 636, 670
clog 880, 882
COBOL 38
Concurrent C 42
const 316, 609, 641
continue 156
cout (выходной поток) 628, 880, 882
CPU (ЦПУ) 33
ctype.h 373
dequeue 545
double 147, 196
E
EBCDIC 389
endl 884
enqueue 545

eofbit 906
escape-код 56, 433
- - \' 433
- - \" 433
- - \? 433
- -\\ 433
- - \a 433
- -\b 433
- - \f 433
- - \n 433
- - \r 433
- - \t 433
- - \v 433
escape-последовательность 433
esc-символ 55
exit 609
EXIT_FAILURE 609
EXIT_SUCCESS 609
F — G
failbit 886
false 68
fclose 496
feof 495
fgetc 493
fgets 493
FIFO (первым пришел —первым ушел) 545
float 105, 107, 196
fopen 495
FORTRAN 38
fprintf 496
fputc 493
fputs 493
fread 504
free 530
friend 636
fscanf 499
fseek 506
fwrite 504
getchar 382, 383
gets 382
get-функция 684
I
inline — встроенная функция 633, 675
int 59, 154
ios::adjustfield 900
ios::basefield 901
ios::fixed 903
ios::floatfield 903
ios::internal 900
ios::scientific 903
ios::showbase 900, 902
ios::showpoint 899
ios::showpos 901
isalnum 373
isalpha 373
iscntrl 373, 376
isdigit 373
isgraph 373, 376
islower 373, 375
isprint 373, 376
ispunct 373, 376
isspace 373, 376
isupper 373, 375
isxdigit 373
L — R
left 899
LIFO (последним пришел — первым ушел) 539
Long 154, 196
lvalue ("значение слева") 162, 699
main 55
make 609
makefile 609
malloc (выделить память) 530
memchr 396, 398
memcinp 396, 398
memcpy 396
memmove 396, 397
memset 396, 399
Pascal 31, 38, 39
pop 540
printf 421
private: 671, 681
protected: 681

public: 671, 681
push 540
putchar 382
puts 382, 383
raise 613
rand 199
RAND_MAX 199
realloc 616
return 190, 193
rewind 499
rvalue ("значение справа") 162
S
scant 434
SEEK_CUR 507
SEEK_END 507
SEEK_SET 507
set-функция 684
short 154
signal 613
signal.h 613
size_t 390
sizeof 324, 530
skipws 898
sprintf 382, 384
srand 201
sscanf 382, 385
stdarg.h 604
stderr (стандартная ошибка) 493
stdin (стандартный ввод) 493
stdio.h 382, 420
stdlib.h 378
stdout (стандартный вывод) 493
strcat 386
strchr 390, 391
strcmp 388
strcpy 386
strcspn 390, 391
strerror 399
string.h 385
strlen 399
strncat 386, 387
strncmp 388
strncpy 386
strpbrk 390, 392
strrchr 390, 392
strspn 390, 393
strstr 390, 394
strtod 378, 379
strtok 391, 394
strtol 378, 380
strtoul 378, 381
struct 453
T— W
template 636, 650
this 636
throw 636
time 203

tmpfile 611
tolower 373, 375
toupper 373, 375
true 68
try 636
typedef 458
union 461
UNIX 36, 39, 41, 151
unsigned 202
uppercase 902
va_arg 604
va_end 604
va_list 604
va_start 604
virtual 636
void 191
void * (указатель на void) 328
volatile 609
width 894, 896
A
абстрактный базовый класс 850
- класс 850
- тип данных (ADT) 670, 739
абстракция 190, 807
- данных 665,739
автоматическая переменная 207
автоматический класс памяти 206
- локальный объект 691

автоматическое приведение аргументов 195
агрегаты 452
активация функции 190
алгоритм 89
альтернатива операторам switch 849, 863
амперсанд (&) 61
анализ данных опроса 269
аппаратная платформа 37
аппаратная часть 31
аргумент 55, 188
аргументы командной строки 606
- функции по умолчанию 644
арифметика указателей 326
арифметические операции 64
- операции присваивания: +=,-=,*=,/= и %= 113
ассоциативность операций 71
- слева направо 66
- справа налево 71
атрибут 665
Б
база данных 492
базовый класс 807
байт 491
безопасная по типу компоновка 647
безопасный по типу ввод/вывод 879
бесконечная отсрочка 355
бесконечный цикл 98, 155
библиотека классов потоков 880
- обработки сигналов 613
- потоков 629
- утилит общего назначения 378
библиотеки классов 808
бинарная операция разрешения областидействия (::) 675
бит 491
битовое поле 472
- - нулевой ширины 471
блок 55, 98, 192
блок default в структуре switch 148, 152
блок памяти 33
блок-схема 90
блочный метод построения программ 37
буква 491
буквенное поле 491
В
ввод/вывод (I/O) 34
вершина 539
виртуальная функция 849
- - базового класса 849
виртуальный деструктор 864
вложенные скобки 66
- структуры 453
- - if/else 95
- управляющие структуры 92, 108
вложенный класс 723
внешняя компоновка 608
внутренняя компоновка 608
возведение 107, 195
восьмеричные числа (начинаются с О) 893,901
восьмеричный формат 421, 422, 434
временная область для обмена значений 269
временный файл 611
вставка отображаемых символов 421
- узла 537, 551
встроенная функция-элемент 679
встроенный тип 758, 760
второе уточнение 103,111
входной блок 33
вхождение в область действия 691
выбор между быстродействием и компактностью 474
выбор 91
вызов по значению 198, 312
- - ссылке 198, 313
- функции 187
вызываемая функция 187
вызывающая функция 187
выравнивание 421
- по левому краю 148, 430

- - правому краю 148, 430
выражение в качестве индекса 249
- с указателями 326
- со смешанными типами 196
выход за пределы массива 257
- из области действия 691
выходной блок 33
вычитание двух указателей 326, 327
- целого из указателя 326, 328
Г— Е
генерация вызова по ссылке 312
- случайных чисел 199
гистограмма 257, 258
глобальная переменная 208, 607
глобальный объект 691
голова очереди 545
данные 32
двоичная цифра 491
двоичное дерево 550
- - поиска 550
двойная косвенная адресация 537
двумерный массив 277
двухместные операции 61, 65
действие 55, 67, 88
декремент 139
- указателя 326, 327
деление на 0 65, 104
- нацело 65
дерево 550
деструктор 674, 691
- базового класса 821
- производного класса 821
десятичная цифра 491
диалоговое вычисление 61
- динамические массивы 615
- объекты 644, 734
- структуры данных 528
динамическое распределение памяти 308
- связывание 850, 862
директива препроцессора 588
-#define 253
длина строки 399
дополнение 464
- до единицы 469
доступ к элементам структуры 455
дружественная функция 727
дружественный класс 727
друзья базового класса 829
- производного класса 814
естественный язык компьютера 35
З
заголовочные файлы стандартной библиотеки 197, 589
заголовочный файл 197, 588, 679
загрузчик 41
задача 34
закрытие файла 496
закрытое наследование 809
закрытый базовый класс 820
заменяющий текст 253, 589
запись 452, 491
- в строчку 65
заполнение 901
заполнитель структуры 472
запуск программы 39
зарезервированные слова 71
защищенное наследование 809
защищенный базовый класс 820
- элемент класса 810, 811
звездочка (*) 64
знак минуса для выравнивания по левому краю 148
- процента % (esc-символ) 61
- равенства = (операция присваивания) 61
значение 61
- переменной 63
- элемента 250
И
идентификатор 59
иерархическое отношение 809
иерархия возведения 196
- данных 491
- классов 810, 851, 852

именованная константа 641 имя 59, 63
- массива 249
- структуры 455
- файла 495
- элемента 453
имя-этикетка 453
- - класса 667
инвертированный набор сканирования 437
индекс 249
- столбца 277
- строки 277
индексация указателя 330
индикатор конца файла 495
инициализатор 644
- базового класса 821
- элемента 687
инициализация 100
- массива 252, 258, 278
- объекта класса 687
- структур 455
- указателей 309
инкапсуляция 666
инкремент указателя 326, 327
интерактивное вычисление 61
интерфейс класса 674, 677
исключение goto 90
исключительная ситуация с плавающей точкой 613
исполнение программы 39
исполняемый образ 41
истинность 68
итератор 743
итерация 213, 219
итоговая сумма 100
К
квадратные скобки 249
клавиша enter 61
- return 61
класс 666, 670
- Array 765
- Date 788
- fstream 882
- if'stream 882
- ios 881
- iostream 881
- istream 881
- ofstream 882
- ostream 881
- String 777
- элемента 827
классы памяти 206
клиент 35
- класса 674, 676
ключ записи 492
- поиска 273
ключевое слово operator 636, 758
- - protected 681
- - virtual 849
ключевые слова 71, 636
код символа 389
команда ее в UNIX 589
комментарий 55, 627
компилятор 36, 39,40
композиция 723
компоновщик 41
компьютер 32
компьютерная программа 32
конвейер 603
"конец ввода данных" 102
конец файла 151, 496, 889, 906
конечное значение управляющей переменной 142
конечный символ 118
конкатенация вызовов функций 762
- - элементов-функций 731
- строк 386
конкретные классы 850 константная элемент-функция 717, 718
константный объект 717
константный указатель 320, 321
- - на константные данные 321
- - на не-константные данные 320
конструирование программного обеспечения 825

конструктор 671, 687, 734
- базового класса 821, 827
- копии 773, 774
- объекта-элемента 727
- по умолчанию 690, 727
- - - базового класса 821
- преобразования 777
- производного класса 821
контейнерные классы 743
контрольное значение 102,139
копирование строк 386
копия значения 198
корневой узел 550
косвенная адресация 309
- ссылка на переменную 309
косвенный базовый класс 820
крах программы 104
круглые скобки () 65, 66
Л
левое выравнивание 899
- поддерево 550
левый потомок 550
лексема 391, 394
линейная структура данных 550
линейный поиск 273
линии перехода 90
листовой узел 550
литерал 55, 371
литеральные символы 421,433
логика switch 849
логическая ошибка 97
логические блоки 33
- операции 158
логическое И (&&) 158, 159
- ИЛИ (||) 158, 159
- отрицание (!) 158, 160
ложность 68
локальная переменная 207, 209
М
макроопределение 590
макрос 589
- с аргументами 590
манипулятор потока 893
- - dec 893
- - Hush 884
- - hex 893
- - oct 893
- - resetiosflags 898
- - setbase 893
- - setfill 901
- - setprecision 894
- - setw 894
- - ws 898 маска 465
маскирование битов 465
массив 248
- п на m 277
- строк 333
- структур 458
- указателей 333
масштабирование 199
масштабируемость 253
машинно-зависимый 35
машинно-независимый 37
машинный язык 35
медиана 269
метка case 148
метод 670
многозадачность 39
многомерный массив 277
множественный выбор 91, 148
моделирование 199, 204
модель действие/решение 56
модель клиент/сервер 35
модульная программа 186
мультипликативные операции 107
мультипрограммирование 34
мультипроцессор 43
"мусор" 100
Н
набор символов 389,491
- - ASCII 150
- сканирования 435
надкласс 809
наиболее вероятное значение 272
направленный ациклический граф 840

нарушение сегментации 613
наследование 806, 807, 849
настройка программного обеспечения 825
научная нотация 423
начальное значение управляющей переменной 139, 142
не равное нулю (true) 68
недопустимая инструкция 613
неименованное битовое поле 472
не-константный указатель на константные данные 318
- - на не-константные данные 317
не-фатальная ошибка 65, 97
нелинейная структура данных 550
неоднозначность в сложном наследовании 834, 838
неопределенное повторение 102, 139
неперегружаемые операции 759
непосредственный базовый класс 820, 849
неразрушающее чтение из ячейки 64
неформатируемый ввод/вывод 892
неявное преобразование 107
неявные преобразования типа 785
нисходящее пошаговое уточнение 33, 102,109,335
номер позиции 249
нотация указатель/смещение 330
нулевой Элемент 249
нули и единицы 490 нуль (false) 68
нуль-символ '\0' 258
О
область действия 206, 209
- - блока 209
- - класса 676, 735
- - прототипа 209
- - символической константы или макроса 591
- - файла 209, 676
- - функции 209
- свободной памяти 643
обработка строк 370, 385
- текстов 370
обратная косая черта \ (esc-символ) 55
обход 551
- с отложенной выборкой 551, 555
- с порядковой выборкой 551, 554
- с предварительной выборкой 551, 554
общая сумма элементов массива 255, 280
объединение строк с другими строками 386
объект 43, 665
объект - элемент данных 723, 727
объектно-ориентированное программирование (ООР) 43,665,807, 826,849
объектный код 40
объявление 59
- класса 670
- массива 251
- структур 453
- функции 195
однострочный комментарий (//) 627
окончание оператора (;) 55
округление 107, 421, 425
окружение (рабочая среда) 39
операнд 61
оператор 55
- goto 90, 616
- присваивания 61
операции как функции 758
- отношения 68, 69
- равенства 68, 69
операция 61
- delete 643, 734
- delete[] 643
- new 643, 734
- взятия адреса & 61, 310
- "взять из" (”) 628
- взятия по модулю (%) 65
- выбора элемента - стрелка 730
- - - точка (.) 730

- - - класса (.) 667
- декремента(--) 114
- дополнения до единицы (-) 469
- извлечения из потока (”) 628, 881
- инкремента(++) 114
- косвенной адресации (*) 309
- левого сдвига (“) 464, 469
- передачи в поток (“) 628, 881
- поразрядного И (&) 464, 465
- - включающего ИЛИ (|) 464, 468
- - исключающего ИЛИ (*) 464, 468
- "послать в" (“) 628
- постдекремента 114
- постинкремента 114
- правого сдвига (”) 464, 469, 470
- предекремента 114
- преинкремента 114
- преобразования 776
- приведения 107
- присваивания (=) 61
- - левого сдвига (“=) 470
- - поразрядного И (&'=) 470
- - - включающего ИЛИ (|=) 470
- - - исключающего ИЛИ ("=) 470
- - правого сдвига (”=) 470
- разрешения области действия :: 675
- разыменования (*) 310
- ссылки & 699
- умножения (*) 64
- элемента структуры - стрелка (->) 455
- - - - точка (.) 455 определение класса 671
- структур 453
- функции 191
определенное повторение 100, 139
оптимизирующий компилятор 207
основной случай рекурсии 212
остановка в узле 555
отказ программы 104
открытие файла 495, 498
открытое наследование 809
открытый базовый класс 820
открытый интерфейс класса 674, 682
отладчик 592
отношение *знает* 828
- *имеет* 808, 827
- *использует* 828
- *является 808,* 827
отображаемый символ 373, 376
отступ 57
очередь 545
ошибка бесконечной рекурсии 816
- времени компиляции 60'
- смещения индекса 250
- - счетчика 142
П
память 33, 63
параметр в определении функции 189, 192
параметр числа записей 506, 508
параметризованный манипулятор потока 880
параметризованный тип 743
параметр-ссылка 637
первичная память 33
первое уточнение 103
переадресация ввода/вывода 602
переадресация потока 421
перегружаемые операции 759
перегруженная дружественная функция-операция 761
- операция != 766, 775
- - [] 771, 775
- - < 785
- - <= 785
- - == 771, 775
- - > 785
- - >= 785
- - присваивания (=) 771, 774
- функция-операция - элемент класса 760
перегрузка двухместной операции 765
- одноместной операции 764
- операций 757

- постфиксной операции 787
- префиксной операции 786
- функций 647
передача массивов в функции 263
- по ссылке 263
- управления 90
переменная 59
- "только для чтения" 641
- класса static 206, 208
- управления циклом 139, 143
- -константа 641
переносимость 37, 42
переопределение элемента базового класса 816
переопределенная виртуальная функция 849,850
перехват 613
перечисление 474
перечислимая константа 474
период хранения 206
персональный компьютер 34
побочные эффекты 198
поведение 665, 670
повторение 98
- управляемое счетчиком 100, 139
повторное использование программного кода 37, 189, 702,826
поддерево 550
подкласс 809
позднее связывание 862
поиск в массиве 273
поле 491
полиморфизм 848, 851,853
пользователь класса 676
поразрядные операции 463
порядок действий 88
последовательная структура 90
последовательное выполнение 90
поток 421
- стандартного ввода 420
- - вывода 420
- стандартной ошибки 421
потоки, определяемые пользователем 881
потоковый ввод 886
- вывод 882
потомки 550
пошаговое уточнение 102
поэлементное копирование 701
правила старшинства операций 65, 66
правило вложения 165
- суперпозиции 163
правое выравнивание 899
- поддерево 550
правый потомок 550
предикатная функция 536, 684
предопределенные потоки 881
символические константы 594
представитель класса 666
представление символа числовым кодом 388
преобразование объекта производного класса в объект базового класса 824
- указателя производного класса в указатель базового класса 825
определяемое пользователем 776
преобразования между встроенными типами и классами 776
- классами различного типа 776
препроцессор С 40, 58, 588
препроцессорная операция конкатенации ## 594
- - преобразования в строку # 594
прерывание 613
приглашение 60
принцип минимума привилегий 207, 316, 320, 679, 717
принятие решения 67
приоритет операций 66
приращение управляющей переменной 139
присваивание структур 454
указателя 309, 310, 328
пробельные символы 71, 92, 376, 898

проверка на выход за пределы массива 783
программа-транслятор 36
программист 32
программное обеспечение 31, 32
производный класс 808, 849
тип 453
произвольный доступ 503
простое наследование 807
условие 158
прототип функции 190, 194
проход сортировки 268
процедурное программирование 666
прямая ссылка на переменную 309
псевдокод 89
псевдослучайные числа 201
пузырьковая сортировка 268
пустой оператор (;) 98
Р
рабочая станция 35
разбиение на функции 33
разбиение строки на лексемы 394
разделение времени 34
разделительные символы 394
"разделяй и властвуй" 186, 189
различение регистра 59
разрушающее считывание в ячейку 63
разыменование указателя 310
рандомизация 201
распределенные вычисления 35
расширение макроса 590
расширяемость 672, 741, 848, 852, 910
реализация класса 677
редактор 39
режим открытия файла 495
- - - a 498
- - - a+ 498
- - - r 498
- - - r+ 498
- - - w 498
- - - w+ 498
рекурсивная функция 210
рекурсивный вызов 212
рекурсия 210, 219
решение 67, 93
родительский узел 551
С
связанный список 308, 531
сдвиг 203, 464
- влево 464, 469
- вправо 464, 470
сервисная функция 684
сиблинги 550
сигнальное значение 102
сигнатура 647
символ 491
- ^ 437
- действия 91
- новой строки (\n) 55, 57
- овала 91
- переадресации ввода < 603
- - вывода > 603
- подавления присваивания (*) 438
- подчеркивания ( ) 59
- присоединения вывода ” 603
- прямоугольника 91
- решения 91
- ромба 91
- стрелки 90
символ-заполнитель 901
- - по умолчанию (пробел) 901
символическая константа 253, 589
символы блок-схемы 90
символьная константа 370
символьное поле 491
синтаксическая ошибка 60, 97
система управления базами данных 492
скаляр 264
скалярная величина 264
сложение указателя с целым 326
сложное наследование 807, 833
смещение 330, 499

событие 613
создание экземпляра объекта 666
сокрытие данных 666, 673
- информации 209
сообщение 55, 665, 670
сортировка 267
- двоичного дерева 554
- погружением 268
- элементов массива 267
составной оператор 97
состояния формата 898
сохранение программы 39
спецификатор класса памяти 206
- - - auto 207
- - - extern 208, 607
- - - register 207
- - - static 208, 608
- преобразования 422
- - % 426, 427, 435
- - c 425, 435
- - d 62, 422, 434
- - e или E 423, 434
- - f 423, 434
- - g или G 424, 434
- - h 422, 434
- - i 422, 434
- - L 424, 435
- - 1 422, 434
- - n 426,427, 435
- - o 422, 434
- - p 263, 426, 427, 435
- - s 201, 425, 435
- - u 422, 434
- - x или X 422, 434
спецификаторы доступа к элементам 671, 727
- преобразования целых 422
спецификация внешней связи 649
- преобразования 61, 69, 71, 421
список аргументов переменной длины 603
- инициализации массива 252
сравнение строк 388
- указателей 326, 329
среднее 269
ссылка на базовый класс 851
стандартизованные компоненты программного обеспечения 808
стандартная библиотека C 37, 56, 187
стандартная ошибка (stderr) 41
стандартный ввод (stdin) 41
- вывод (stdout) 41
- заголовочный файл <iomanip.h> 880
- - - ввода/вывода 59 статическая элемент-функция 736
статический локальный объект 691
- период хранения 206, 208
- элемент данных 735
статическое связывание 850
стек 539
столбцовая диаграмма 257
строка 55, 257, 259
- поиска 394
- символов 55
- управления форматом 421, 434
строковая константа 371
строковый литерал 371
структура FILE 493, 495, 497
- выбора if 93
- - if/else 94
- - switch 148
- повторения do/while 154
- - for 141
- - while 98, 154
- с двойным выбором 91
- с единичным выбором 91
- со множественным выбором 91
- ссылающаяся на себя 453, 529
структурное программирование 39, 54
структуры выбора 90
- повторения 90, 139
суперкомпьютер 32
суперпозиция управляющих структур 92
суффикс float (f или F) 611

- long double (1 или L) 611
- long int (l или L) 610
- unsigned int (u или U) 610
- unsigned long (ul или UL) 610
- -амперсанд (&) 637
счетчик 100
- цикла 139
Т
таблица виртуальных функций 863
- значений 277
- истинности 159
табличная форма 251
тело функции 55
- цикла 98, 154
терминал 34
тернарная операция 95
тильда (-) в имени деструктора 674
тип возвращаемого значения 191
- данных 669
- компоновки 206
- переменной 63
- структуры 453
- определяемый пользователем 666, 757
типы ссылки 637
- указателей 309, 328
точка с запятой ; (окончание оператора) 55
точность 107, 429
- по умолчанию 107, 423, 894
У
удаление узла 533, 538, 539
узел 531
узел-потомок 550
указатель 308
- NULL 309, 530
- this 729
- базового класса 811
- на void (void *) 328
- - абстрактный класс 852
- - базовый класс 850
- - объект базового класса 825
- - объект производного класса 825
- - производный класс 852
- - символ 333,337
- - структуру 453, 456
- - указатель 537
- - функцию 339
- позиции файла 499
- производного класса 811
- файла 495
унарная (одноместная) операция 160
- операция разрешения области действия (::) 681, 807
управление доступом к элементам 681, 807
управление, поток управления 72
- программой 89
управляющая строка формата 61, 62
- структура 90
- - if 68
управляющие структуры с одним входом / одним выходом 92, 163
управляющий символ 376
усечение 107
ускоренная разработка прикладных программ (RAD) 702
условие 68
- завершения 100
- продолжения цикла 143, 154
условная компиляция 592
- операция (?:) 95
условное выполнение директив препроцессора 588
услуги, предоставляемые классом 682
устройство ввода 33
- вывода 33
Ф
фаза завершения 105
- инициализации 105
- обработки 105
файл 491
- исходного кода 679
- последовательного доступа 492
- произвольного доступа 503
файловый сервер 35

фатальная ошибка 65, 97
фигурные скобки {} 55, 97
фиктивное значение 102 флаг 421, 430
- пробел 430
- - (минус) 430
-# 431
- + (плюс) 430
- 0 (нуль) 431
флаги формата 898
флаговое значение 102
формат целого без знака 422
- - со знаком 422
форматирование в памяти 880
форматированный ввод/вывод 500, 880
функции математической библиотеки 188
- преобразования строк 377
- сравнения строк 388
функция 37, 56,187
- getchar 150
- pow 147
- printf 58, 62
- scant 58, 61
- доступа 684
- факториала 213
- определяемая программистом 187
- -элемент 666, 670
- - bad 906
- - clear 906
- - eof 906
--fail 906
- - fill 901
- - flags 896, 898
- - gcount 892
- - get 889
- - getline 890
- - good 906
- - ignore 891
- - operator void* 907
- - operator! 907
- - peek 891
- - precision 984
- - put 886
- - putback 891
- - rdstate 906
- - read 892
- - setf 896, 898
- - tie 910
- - unsetf 896, 898
- - write 892
Х—Я
хвост очереди 545
целое 59
- типа long 422, 434
- типа short 422, 434
- число 500
целочисленное деление 107
целостность (корректность) данных 687, 688,694
центральное процессорное устройство (CPU) 33
цикл 92, 139
число с плавающей точкой 105, 423, 434
чисто виртуальная функция (=0) 851
шаблон 649
- класса 743
- функции 649
шестнадцатеричные цифры 373
- числа (начинаются с Ох или ОХ) 901
шестнадцатеричный формат 421
ширина битового поля 471
- поля 148, 428
экран 33, 41
экспоненциальный формат с плавающей точкой 423
элемент 453,461
- данных 666, 670
- массива 249
- случайности 199
этикетка структуры 453, 667
явное преобразование 107
- - типа (операция приведения) 776

язык высокого уровня 36, 38
- программирования 35
ясность 42
ячейка памяти 63

Предисловие

Эту книгу написали два человека — старый и молодой. Старик программирует и учит программировать других больше 30-ти лет. Молодой человек работал программистом около десяти лет, пока им не овладела страсть к учительству и писательству. Старик программирует и учит благодаря своему опыту. Молодой человек — благодаря неистощимому запасу энергии. Старик стремится к ясности; молодой человек — к наивысшей эффективности. Старику нравится красота и элегантность. Молодому человеку хочется результатов. Мы объединились, чтобы на свет появилась книга, которую, как мы надеемся, вы найдете полезной, интересной и занимательной.

В большинстве учебных курсов языку С учат людей, которые знают, как программировать. Многие преподаватели думают, что сложность С и ряд других трудностей делают этот язык непригодным для начального обучения программированию — как раз для того, на что нацелена эта книга. Так почему же мы ее написали?

Язык С занял особое место среди языков реализации систем в компьютерной индустрии, и есть все основания верить, что его объектно-ориентированный вариант С++ станет ведущим языком середины — конца 90-х годов. Харви Дейтел в течение 13-ти лет преподавал в университетах Паскаль, делая акцент на разработке ясно написанных, хорошо структурированных программ. Большинство из того, чему учат во вводных курсах по Паскалю — это основные принципы структурного программирования. Мы представили этот материал в точности так, как это делал старший из нас в своих университетских курсах. Тут имеются определенные ловушки, но там, где они встречаются, мы указываем на них и объясняем, как можно их избежать. Исходя из опыта мы можем сказать, что обучающиеся справляются с данным курсом ничуть не хуже, чем с аналогичным, но ориентированным на Паскаль. Однако здесь есть одно существенное отличие: у студентов появляется очень сильный стимул благодаря тому, что они учат язык, который немедленно окажется им необходим, как только они покинут стены университета. Они с энтузиазмом относятся к работе с материалом — что немаловажно, учитывая повышенную трудность изучения С.

Наша цель была ясна: написать учебник по программированию на С для вводных курсов университетского уровня с ориентацией на студентов, имеющих небольшой или вообще нулевой опыт программирования, но одновременно и книгу, предлагающую достаточно строгое изложение теории и практики, характерное для традиционных курсов по языку С. Чтобы достичь этих целей, мы написали книгу более объемистую, чем обычные учебники, — потому что наш текст, помимо всего прочего, постепенно приучает читателя к принципам структурного программирования. Около 1000 студентов изучали предлагаемый здесь материал на наших курсах. Десятки тысяч учащихся по всему миру изучали С по первому изданию этой книги.

Книга содержит большой набор примеров, упражнений и проектов из различных областей, чтобы дать читателю возможность решать интересные задачи из реального мира.

Книга сосредоточивает внимание на принципах качественного конструирования программного обеспечения и делает акцент на ясности программ, написанных с использованием методологии структурного программирования. Мы пытаемся избегать эзотерической терминологии и спецификаций синтаксиса в пользу обучения на примерах.

Среди педагогических элементов, применяемых в тексте — законченные программы и примеры их вывода, демонстрирующие обсуждаемые концепции; краткое содержание, приведенное в начале каждой главы; распространенные ошибки и правила хорошего стиля программирования, сообщаемые по ходу изложения и сведенные в отдельный раздел в конце главы; резюме и контрольные вопросы с ответами к ним; и наконец, богатейший набор упражнений, какой вы вряд ли найдете в других книгах по С. Упражнения варьируются от простых вопросов на повторение до больших программных задач и настоящих полноценных проектов. Преподавателям, которым требуются достаточно сложные проекты в качестве курсовых работ для студентов, найдут много подходящих задач, приведенных в упражнениях в главах 5-й по 21-ю. Мы приложили немалые усилия к тому, чтобы благодаря упражнениям данный курс приобрел для студента большую ценность. Программы в тексте книги тестировались при помощи совместимых с ANSI компиляторов на Sun SPARCstation, Apple Macintosh (Think C), IBM PC (Turbo C, Turbo C++ и Borland C++) и DEC VAX VMS (VAX C).

Данная книга следует стандарту ANSI C. Многие особенности ANSI C не реализуются компиляторами для более ранних версий С. Обратитесь к руководствам по вашей конкретной системе, чтобы выяснить больше подробностей относительно языка, либо достаньте документ ANSI/ISO 9899: 1990, «American National Standard for Information Systems — Programming Language — C».

Об этой книге

Все компоненты и отличительные особенности этой книги направлены на то, чтобы помочь студентам в изучении языка.

Цели урока

Каждая глава начинается с формулирования учебных задач. Благодаря этому студент получает представление о том, к чему следует подготовиться, а также, по прочтении главы, он может оценить, насколько ему удалось выполнить данные задачи.

Содержание глав

Содержание в начале каждой главы помогает студенту подойти к изучению материала, так сказать, «сверху вниз». Тем самым он может видеть, что ждет его впереди, и спланировать свой темп работы.

Разделы

Каждая глава организована в виде небольших разделов, посвященных ключевым моментам темы. Каждый аспект С представлен в контексте законченной работающей программы. После каждой программы приводится «окно», содержащее вывод, полученный при ее запуске. Это позволит студенту убедиться, что программа работает, как ожидалось. Соотнесение вывода с операторами программы, которые его производят, является превосходным подспорьем в изучении и закреплении принципиального материала. Наши программы разработаны для того, чтобы испытать разнообразие возможностей С. Внимательное чтение книги должно выглядеть так, как будто эти программы действительно вводятся, компилируются и запускаются на компьютере.

Иллюстрации

Книга содержит разнообразные рисунки и диаграммы. Обсуждение структурных блок-схем, помогающее оценить значение управляющих структур и структурного программирования, сопровождается ясными диаграммами. Глава о структурах данных содержит массу рисунков, иллюстрирующих создание и поддержку связанных списков, очередей, стеков и двоичных деревьев.

Вспомогательные элементы оформления

Мы используем пять элементов оформления, чтобы помочь студентам сконцентрировать внимание на важных аспектах разработки программ, их тестирования и отладки, вопросах эффективности и переносимости. Мы выделяем соответствующую информацию под рубриками *Хороший стиль программирования, Распространенные ошибки программирования, Советы по повышению эффективности, Советы по переносимости кода* и *Общие методические замечания.*

Хороший стиль программирования

В тексте выделяются советы относительно целесообразного стиля программирования. Они привлекают внимание читателя к приемам, помогающим писать лучшие программы. Эти советы сообщают самое ценное из того, что мы вынесли из своего сорокалетнего (в сумме) опыта.

Распространенные ошибки программирования

Студенты, изучающие язык — особенно те, кто проходит начальный курс обучения, — склонны совершать одни и те же ошибки. Привлечение внимания к этим распространенным ошибкам программистов очень помогает в этом отношении. Лучше учиться на чужих ошибках, чем на своих — это позволяет избежать напрасной потери времени.

Советы по повышению эффективности

Мы считаем, что написание ясных и понятных программ — важнейшая из целей начального курса по программированию. Однако студенты хотят писать самые быстрые, самые компактные, самые экономные, либо выдающиеся в каком-либо ином отношении программы. Им действительно небезразлична эффективность программ. Они хотят знать, как можно оснастить свои программы каким-нибудь «усилителем». Поэтому мы включили в текст *Советы по повышению эффективности*, чтобы подчеркнуть возможные пути улучшения программ.

Советы по переносимости программ

Разработка программного обеспечения — сложное и чрезвычайно дорогое занятие. Организации, разрабатывающие программы, часто вынуждены производить различные их версии, приспособленные к разнообразным существующим компьютерам и операционным системам. Поэтому сегодня делается сильный акцент на переносимости, т.е. на производстве такого программного обеспечения, которое будет работать без изменений на многих компьютерных системах. Многие утверждают, что C — лучший язык для разработки переносимого программного обеспечения. Некоторые полагают, что если они реализуют свою программу на C, она автоматически будет переносимой. Это попросту ошибочно. Достижение переносимости требует тщательного и осмотрительного проектирования. Здесь масса ловушек. В самом документе ANSI по

Стандартному С перечень потенциальных трудностей занимает 11 страниц. Мы включили в текст многочисленные *Советы по переносимости кода*, в которых обобщили свой собственный опыт разработки переносимого программного обеспечения, дополнив его внимательным изучением раздела стандарта ANSI, посвященного переносимости.

Общие методические замечания

Этот элемент оформления — новый во втором издании. Мы суммировали под этой рубрикой замечания, касающиеся архитектуры и конструирования программных систем, особенно больших систем.

Резюме

Каждая из глав оканчивается набором дополнительных разделов, призванных служить определенным педагогическим целям. Мы приводим детальное резюме главы в форме перечня ключевых пунктов. Это помогает студенту пересмотреть и закрепить в памяти важнейшие моменты материала.

Терминология

Раздел *Терминология* содержит список важнейших терминов (в алфавитном порядке), определяемых в данной главе. Цель его та же — закрепление пройденного материала. Затем мы приводим все имеющиеся в главе вставки, собранные по рубрикам — *Хороший стиль программирования*, *Распространенные ошибки программирования*, *Советы по повышению эффективности*, *Советы по переносимости кода* и *Общие методические замечания*.

Упражнения для самоконтроля

Эти многочисленные упражнения, с исчерпывающими ответами на них, включены в книгу для самостоятельного изучения. Они позволяют студенту приобрести уверенность в применении изученного и подготовиться к тому, чтобы попробовать свои силы в решении настоящих задач.

Упражнения

Завершает каждую главу объемистый набор упражнений, диапазон которых меняется от простого повторения важнейших терминов и понятий, через написание одиночных операторов С, небольших фрагментов функций, законченных функций и программ — до написания больших курсовых проектов. Такое большое число упражнений позволяет преподавателям приспособить свои курсы к специфическим потребностям аудитории и менять учебные задания от семестра к семестру.

Они могут использовать эти упражнения в качестве домашних заданий, для коротких опросов и для экзаменационных сессий.

Обзор книги

Книгу можно разделить на три основных части. Первая из них, охватывающая главы с 1-й по 14-ю, содержит очень подробное изложение языка С, включая формальное введение в структурное программирование. Вторая часть — что отличает нашу книгу от других учебников по С, — главы с 15-й по 21-ю, представляет собой серьезное введение в язык С++ и объектно-ориентированное программирование, пригодное в качестве учебного пособия для сту-

дентов старших курсов колледжа. Третья часть, приложения А — Д, содержат разнообразный справочный материал, дополняющий основной текст.

Глава 1, «Введение», рассказывает о том, что такое компьютеры, как они работают и как их программируют. В ней вводится понятие структурного программирования и объясняется, почему переход к структурным методикам явился своего рода революцией в практике написания программ. В главе дается краткая история развития языков программирования от машинных языков через языки ассемблера — к языкам высокого уровня. Рассказано об истоках языка С. В главе также представлено введение в среду программирования С.

Глава 2, «Введение в программирование на С», дает общее представление о том, как пишутся программы на С. Приводится подробное описание принятия решений и арифметических операций. После прочтения главы учащийся будет понимать, как написать простую, но тем не менее законченную программу на С.

Глава 3, «Структурное программирование», является, возможно, наиболее важной во всей книге, особенно для студента, всерьез занимающегося компьютерными дисциплинами. В ней вводится понятие алгоритмов (процедур) решения задач. Объясняется важность структурных методов для написания понятных, простых в отладке и сопровождении программ, которые с большей вероятностью могут правильно заработать с первой попытки. Вводятся фундаментальные управляющие конструкции структурного программирования, а именно последовательная, выбора (**if** и **if/else**) и повторения (**while**). Объясняется методика нисходящего пошагового уточнения, критическая для написания хорошо структурированных программ. Обсуждается популярный инструмент проектирования программ — структурный псевдокод. Методы и подходы, изложенные в 3-й главе, приложимы к структурному программированию на любом языке, не только на С. Глава помогает учащемуся выработать «хорошие привычки» в программировании, заранее подготавливая его к более серьезным программным задачам в оставшейся части текста.

Глава 4, «Управление программой», уточняет понятия структурного программирования и представляет дополнительные управляющие структуры. Детально исследуется повторение и сравниваются альтернативы циклов, управляемых счетчиком, и циклов, управляемых контрольным значением. Вводится структура **for** как удобное средство реализации циклов с управлением счетчиком. Представляются структура выбора **switch** и структура повторения **do/while**. Заканчивает главу обсуждение логических операций.

Глава 5, «Функции», обсуждает проектирование и построение программных модулей. В С имеются стандартные библиотечные функции и функции, определяемые программистом, возможности рекурсии и вызова по значению. Методики, представленные в 5-й главе, существенны для написания и понимания преимуществ правильно структурированных программ, особенно больших программ и вообще того рода программного обеспечения, которое системным программистам скорее всего придется разрабатывать для приложений реального мира. В качестве эффективного средства для решения сложных задач предлагается стратегия «разделяй и властвуй»; функции позволяют программисту разбить сложную программу на более простые взаимодействующие компоненты. Студентам нравится заниматься случайными числами и моделированием, поэтому их увлекает раздел главы, обсуждающий игру в кости, где изящно используются управляющие структуры. Глава также предлагает основательное введение в рекурсию; приводится таблица, где перечислены приме-

ры и упражнения на рекурсию (всего их 31), встречающиеся на протяжении остальной части книги. В некоторых учебниках рекурсия вводится в какой-либо из последних глав; но мы считаем, что эту тему лучше всего охватывать постепенно, на всем протяжении текста. В объемистый набор из 39 упражнений в конце 5-й главы включены некоторые классические рекурсивные задачи, например, Ханойская башня.

Глава 6, «Массивы», обсуждает структурирование данных в виде массивов, т.е. групп логически связанных элементов одного типа. В главе представлены многочисленные примеры как одномерных, так и двумерных массивов. Структурирование данных повсеместно признается не менее важным моментом разработки структурных программ, чем применение управляющих структур. Примеры в главе исследуют различные распространенные операции над массивами, печать гистограмм, сортировку данных, передачу массивов функциям и дают представление об анализе наблюдаемых данных. Следует отметить подробное рассмотрение метода двоичного поиска, как важнейшего шага вперед по сравнению с линейным поиском. Упражнения в конце главы включают в себя особенно много интересных и трудных задач. Сюда относятся улучшение методов сортировки, проектирование системы резервирования авиабилетов, введение в концепцию «черепаховой графики» (ставшую знаменитой в качестве языка LOGO), а также задачи об обходе доски конем и о восьми ферзях, где вводятся понятия эвристического программирования, столь широко используемые в области исследования искусственного интеллекта.

Глава 7, «Указатели», представляет одну из самых мощных особенностей языка С. Глава в деталях обсуждает операции, применяемые с указателями, вызов по ссылке, выражения с указателями, арифметику указателей, взаимосвязь указателей и массивов, массивы указателей и указатели на функции. В число упражнений главы входит моделирование классического соревнования между зайцем и черепахой, а также алгоритмы тасования и сдачи карт. Имеется специальный раздел, озаглавленный «Как построить свой собственный компьютер». В нем объясняются принципы программирования на машинном языке и представлен проект, включающий в себя проектирование и реализацию компьютерного симулятора, позволяющего читателю писать и запускать программы на «машинном» языке. Этот раздел будет особенно интересен для тех, кто хочет понять, как на самом деле работает компьютер. Нашим студентам очень нравится этот проект, и они часто предлагают и реализуют существенные его усовершенствования; некоторые из них предлагаются читателю в качестве упражнений. Другой специальный раздел имеется в главе 12, где читателю демонстрируется создание компилятора; машинный код, генерируемый компилятором, затем исполняется на симуляторе, разработанном в 7-й главе.

В главе 8, «Символы и строки», речь идет об основных принципах обработки нечисловых данных. В главу включен подробный обзор функций обработки символов и строк, имеющихся в библиотеках С. Методики, обсуждаемые здесь, широко используются при создании текстовых редакторов, типографских программ макетирования и приложений обработки текстов. Глава завершается интересным набором из 33 упражнений, исследующих обработку текстов. Читателю понравятся задачи, где требуется писать лимерики и стохастические стихотворения, переводить английский текст на «свинячью латынь», генерировать семибуквенные слова, эквивалентные заданному номеру телефона, осуществлять выравнивание текста, защищать чеки, записывать сумму чека прописью, генерировать код Морзе, делать метрические преобра-

зования и отправлять угрожающие письма с напоминаниями об уплате. Наконец, последнее упражнение предлагает даже использовать компьютерный словарь для того, чтобы генерировать кроссворды!

Глава 9, «Форматированный ввод/вывод», рассатривает мощные возможности функций **printf** и **scanf**. Мы обсуждаем различные детали форматирования вывода с помощью **printf**, такие как округление значений с плавающей точкой до заданного количества значащих цифр, выравнивание столбцов чисел, левое и правое выравнивание в поле, вставка литеральной информации, принудительный вывод знака «плюс», печать начальных нулей, использование экспоненциальной нотации, восьмеричных и шестнадцатеричных чисел и управление шириной поля и точностью представления. Приводятся все esc-последовательности функции **printf** для перемещения курсора, вывода специальных символов и подачи звукового сигнала. Затем исследуются все возможности форматного ввода с помощью функции **scanf**, включая ввод специфических типов данных и пропуск символов во входном потоке. Мы обсуждаем все спецификаторы формата **scanf** для чтения десятичных, восьмеричных, шестнадцатеричных значений, чисел с плавающей точкой, символов и строк, а также для сканирования ввода на предмет поиска совпадений (или несовпадений) с заданным набором символов. Упражнения главы демонстрируют практически все возможности форматирования в C.

Глава 10, «Структуры, объединения, операции с битами и перечисления» представляет широкое разнообразие важных элементов языка. Структуры напоминают «записи» в Паскале и других языках программирования — они группируют элементы данных различных типов. В главе 11 структуры используются для формирования файлов, состоящих из информационных записей. В главе 12 структуры в соединении с указателями и динамическим распределением памяти применяются для создания динамических структур данных, таких как связанные списки, очереди, стеки и деревья. Объединения позволяют использовать область памяти в разные моменты времени для данных различных типов. Такое разделение памяти позволяет сократить требования программы к доступному объему памяти или к пространству на устройствах вторичного хранения данных. Перечисления представляют собой средство определения удобных в использовании символических констант; такие константы помогают сделать программу самодокументированной. Мощные возможности C в плане манипуляции отдельными битами позволяют программистам писать код, использующий низкоуровневые особенности оборудования. Программы могут обрабатывать битовые последовательности, устанавливать или сбрасывать отдельные биты и хранить информацию более компактно. Эти черты C, свойственные обычно только языкам ассемблера, чрезвычайно ценятся программистами, пишущими системное обеспечение, например, операционные системы или сетевое программное обеспечение. Главным примером этой главы является пересмотренная, высокоэффективная модель тасования и сдачи карт. Он дает преподавателю прекрасную возможность продемонстрировать, что такое качество алгоритма.

Глава 11, «Обработка файлов», обсуждает методы обработки текстовых файлов в последовательном и произвольном режимах доступа. Глава начинается с представления иерархии данных от отдельных битов к байтам, к полям, к записям и, наконец, к файлам. Затем показан простой подход языка C к файлам и потокам. Последовательный доступ к файлам обсуждается в контексте трех программ, которые показывают, как файлы открываются и закрываются, как производится запись в последовательный файл и затем чтение из

него. Произвольный доступ к файлам обсуждается на примере четырех программ, показывающих, как, используя последовательный доступ, создать файл для дальнейшей работы с произвольным доступом к нему, и как производится последовательное чтение файла с произвольным доступом. Четвертый пример для демонстрации произвольного доступа комбинирует различные приемы как последовательного, так и произвольного доступа к файлам, являясь законченной программой обработки транзакций. Студенты на наших промышленных семинарах говорили, что после изучения материала по обработке файлов они смогли писать серьезные программы, сразу оказывавшиеся полезными в работе их организаций.

Глава 12, «Структуры данных», рассматривает создание динамических структур. Глава начинается с представления структур, ссылающихся на себя, и динамического распределения памяти. Затем обсуждается, как создавать и поддерживать различные динамические структуры данных, включая связанные списки, очереди, стеки и деревья. Для каждого типа структур мы приводим законченные работающие программы и показываем, что выводится в результате их работы. Глава 12 помогает студенту в совершенстве овладеть приемами работы с указателями. В главе имеется масса примеров с косвенной адресацией и двойной косвенной адресацией — особенно трудной конструкцией. Одной из проблем работы с указателями является то, что студентам трудно визуально представить себе структуры данных и связи их узлов друг с другом. Поэтому мы приводим иллюстрации, показывающие эти связи и последовательность, в которой они создаются. Пример с двоичным деревом завершает изучение указателей и динамических структур данных. Этот пример создает двоичное дерево, исключая при этом дублирование данных, и производит рекурсивные обходы дерева с предварительной, порядковой и отложенной выборкой. Студенты получают настоящее удовольствие от изучения и реализации этого примера. Особенное впечатление на них производит тот факт, что обход дерева с порядковой выборкой печатает значения узлов в сортированном виде. Завершает главу большое собрание упражнений. Важнейшие из них объединяются в специальном разделе «Как построить свой собственный компилятор». Упражнения проводят студента через процесс разработки программы преобразования инфиксной нотации в постфиксную и программы постфиксной оценки. Затем мы модифицируем алгоритм постфиксной оценки так, чтобы он генерировал код машинного языка. Компилятор помещает этот код в файл (используя методы главы 11). Затем студенты запускают машинный код, генерируемый их компилятором, на программном симуляторе, построенном в упражнениях 7-й главы.

Глава 13, «Препроцессор», дает подробные описания директив препроцессора. Включена более полная информация по директиве **#include**, которая приводит к включению перед компиляцией копии указанного файла вместо директивы, и директиве **#define**, создающей символические константы и макросы. В главе обсуждается условная компиляция, позволяющая программисту управлять исполнением препроцессорных директив и компиляцией кода программы. Также описывается операция #, преобразующая свой операнд в строку, и операция ##, выполняющая конкатенацию двух лексем. Представлены пять предопределенных символических констант (**__LINE__**, **__FILE__**, **__DATE__**, **__TIME__** и **__STDC__**). Наконец, обсуждается макрос **assert** из заголовочного файла **assert.h**. Этот макрос является ценным инструментом тестирования, отладки и верификации программ.

Глава 14, «Специальные вопросы», посвящена довольно сложной тематике, которая обычно не затрагивается во вводных курсах по C. Раздел 14.2 показывает, как можно переадресовать ввод программы на файл и направить в файл ее вывод, направить вывод одной программы на вход другой (конвейер), а также присоединить вывод программы в конец существующего файла. Раздел 14.3 описывает разработку функций со списками аргументов переменной длины. Раздел 14.4 показывает, как передать функции **main** аргументы командной строки. Раздел 14.5 говорит о компиляции программ, компоненты которых распределены по различным файлам. Раздел 14.6 обсуждает регистрацию с помощью **atexit** функций, которые должны вызываться при завершении программы, а также завершение исполнения программ вызовом функции **exit**. В разделе 14.7 описываются модификаторы типа **const** и **volatile**. Раздел 14.8 показывает, как специфицируется тип числовых констант посредством суффиксов — целых и с плавающей запятой. Раздел 14.9 обсуждает двоичные файлы и использование временных файлов. В разделе 14.10 показано, как использовать библиотеку обработки сигналов, чтобы перехватывать непредвиденные события. Наконец, раздел 14.11 посвящен созданию и использованию динамических массивов с помощью функций **calloc** и **realloc**.

В первое издание книги мы включили одну главу, являвшуюся введением в C++ и объектно-ориентированное программирование. За время, прошедшее с тех пор, многие университеты решили дополнить свои курсы по C введением в C++. Поэтому в новом издании мы расширили наше изложение C++ и объектно-ориентированного программирования до семи глав; их материала достаточно для курса продолжительностью в один семестр.

Глава 15, «C++ как улучшенный C», представляет те новые элементы языка, которые не являются объектно-ориентированными. Они направлены на совершенствование процесса написания обычных, процедурно-ориентированных программ. В главе обсуждаются однострочные комментарии, потоковый ввод/вывод, объявления, создание новых типов данных, прототипы функций и проверки типа, встроенные функции (как альтернатива макросам), параметры-ссылки, модификатор **const**, динамическое распределение памяти, аргументы по умолчанию, унарная операция разрешения области действия, перегрузка функций, спецификации внешней связи и шаблоны функций.

Глава 16, «Классы и абстракция данных», дает читателю превосходную возможность изучить абстракцию данных «правильно» — при посредстве языка (C++), специально предназначенного для реализации абстрактных типов данных (ADT). В последние годы абстракция данных стала главной темой вводных курсов по программированию, использующих Паскаль. Когда мы писали книгу, то рассматривали возможность изложения абстракции данных на базе языка C, но вместо этого решили включить в нее подробное введение в C++. Главы 16, 17 и 18 содержат серьезный материал по абстракции данных. В главе 16 обсуждается реализация ADT как **struct** и как классов в стиле C++, доступ к элементам класса, отделение интерфейса от реализации, использование функций доступа и сервисных функций, инициализация объектов с помощью конструкторов и их уничтожение с помощью деструкторов, присваивание путем поэлементного копирования, а также повторное использование программного обеспечения.

Глава 17, «Классы: часть II» продолжает изучение классов и абстракции данных. В ней обсуждается создание и использование константных объектов, константных функций-элементов, композиция — построение классов, включающих в качестве элементов объекты других классов, дружественные функ-

ции и классы, имеющие специальные права доступа к закрытым элементам классов, указатель **this**, позволяющий объекту знать свой собственный адрес, динамическое распределение памяти, статические элементы класса, содержащие единую для всего класса информацию, контейнерные классы, итераторы, и шаблоны классов. Шаблоны классов — одно из недавних дополнений, появившихся в результате эволюции языка C++. Они позволяют программисту выразить самую сущность абстрактного типа данных (например, стека, массива или очереди) и затем создавать — с минимальным количеством дополнительного кода — версии этого ADT для конкретных типов (таких, как стек **int**, стек **float** или очередь **int** и т.д.). По этой причине шаблоны классов часто называют параметризованными типами.

Глава 18, «Перегрузка операций», посвящена одному из наиболее популярных разделов в курсах по C++. Студенты изучают ее с удовольствием. Они находят, что материал главы прекрасно соответствует изложению абстрактных типов данных в главах 16 и 17. Перегрузка операций дает программисту возможность сообщить компилятору, как следует использовать существующие операции с объектами новых типов. C++ заранее знает, как эти операции применяются к объектам встроенных типов, таких, как целые, числа с плавающей точкой и символы. Но допустим, что мы создаем новый класс строк. Что для него должен означать знак «плюс»? Многие программисты используют плюс со строками, подразумевая их конкатенацию. В этой главе читатель узнает, как «перегрузить» данный знак таким образом, что если он будет стоять между двумя строками в выражении, компилятор генерирует вызов «функции-операции», которая выполнит конкатенацию этих строк. В главе рассматриваются основы перегрузки операций, ограничения на перегрузку, перегрузка операций функциями-элементами класса и дружественными функциями, перегрузка одноместных и двуместных операций и операции преобразования одного типа в другой. Особенностью главы является большое число развернутых примеров, а именно примеров класса массивов, класса строк, класса даты, класса сверхдлинных целых и класса комплексных чисел (последние два относятся к упражнениям).

Глава 19, «Наследование», имеет дело с одной из фундаментальных черт всех объектно-ориентированных языков. Наследование является формой повторного использования программного кода, когда новые классы можно быстро получить, взяв свойства существующего класса и дополнив их необходимыми новыми возможностями. В главе рассматриваются понятия базовых и производных классов, защищенные элементы, открытое, защищенное и закрытое наследование, непосредственные и косвенные базовые классы, использование конструкторов и деструкторов в базовых и производных классах и значение наследования в конструировании программного обеспечения. Наследование (отношение «является») сравнивается с композицией (отношение «имеет»), а также вводятся понятия отношений «использует» и «знает». В главе приводится несколько развернутых примеров. В частности, один из них реализует иерархию классов точек, кругов и цилиндров. Заключает главу пример сложного наследования — особенности языка C++, позволяющей сформировать производный класс путем наследования атрибутов и поведения нескольких базовых классов.

Глава 20, «Виртуальные функции и полиморфизм», посвящена другому фундаментальному аспекту объектно-ориентированного программирования, а именно полиморфному поведению. Когда многие классы связаны друг с другом через наследование одному и тому же базовому классу, объект любого

из производных классов может рассматриваться как объект базового класса. Это позволяет писать программы в довольно общем виде, не зависящем от конкретного типа объектов производных классов. Новые виды объектов могут обрабатываться той же самой программой, что делает программные системы расширяемыми. Полиморфизм позволяет устранить из программ сложную логику переключателей и сделать код более «прямолинейным». Менеджер экрана игровой программы может, например, просто послать сообщение типа «нарисовать» каждому объекту в связанном списке визуальных объектов. Каждый объект знает, как себя нарисовать. В программу можно ввести новый объект, не модифицируя ее, если новый объект тоже умеет нарисовать себя. Такой способ программирования чаще всего используется при разработке чрезвычайно популярных сегодня графических интерфейсов пользователя. В главе рассматривается механизм реализации полиморфного поведения посредством виртуальных функций. Проводится различение абстрактных классов (которые не могут иметь представителей) и конкретных классов (объекты-представители которых можно создавать). Абстрактные классы полезны при создании наследуемого интерфейса для всей иерархии классов. В главе разбираются два больших примера на полиморфизм — система начисления заработной платы и вторая версия иерархии точек, кругов и цилиндров, рассмотренной в главе 19.

Глава 21, «Потоки ввода/вывода в C++», содержит чрезвычайно подробное описание нового, в объектно-ориентированном стиле, ввода/вывода C++. Многие курсы по C проводятся с использованием компиляторов C++, и преподаватели часто предпочитают учить вводу/выводу в новом стиле, не пользуясь старыми функциями **printf/scanf**. В главе рассматриваются различные аспекты ввода/вывода C++, куда входит вывод с применением операции помещения в поток и ввод с применением операции извлечения из потока, безопасный по типу ввод/вывод (большое улучшение по сравнению с C), форматируемый и не форматируемый (в целях эффективности) ввод/вывод, манипуляторы потока для управления основанием (десятичное, восьмеричное или шестнадцатеричное основание), представлением чисел с плавающей точкой и шириной поля вывода; определяемые пользователем манипуляторы, состояния ошибки потока, ввод/вывод объектов определяемого пользователем типа и привязка выходных потоков ко входным (для того, чтобы запросы действительно появлялись на экране перед тем, как пользователь должен дать на них ответ).

В нескольких приложениях приводится ценный справочный материал. В *частности, в приложении А мы представляем сводку синтаксиса C; сводка* всех функций стандартных библиотек C приводится в приложении Б; таблица приоритетов и ассоциации операций дана в приложении В; приложение Г содержит таблицу ASCII-кодов символов; и, наконец, в приложении Д проводится обсуждение двоичной, десятичной, восьмеричной и шестнадцатеричной систем счисления. В основу приложения Б положен, с письменного разрешения Американского института национальных стандартов, документ стандарта ANSI; приложение является детальным, очень ценным для практического программиста материалом. Приложение Д является учебным пособием по системам счисления, в которое входят контрольные вопросы и упражнения.

Мы в полной мере берем на себя ответственность за недостатки и неточности, которые могут встретиться в нашей книге. Мы с благодарностью примем ваши замечания, критику, исправления и предложения по улучшению текста. Пожалуйста, *шлите нам свои предложения и дополнения к рубрикам Хороший стиль программирования, Распространенные ошибки программирова-*

ния, Советы по повышению эффективности, Советы по переносимости кода и *Общие методические замечания.*

Направляйте всю корреспонденцию на наш адрес электронной почты:

`deitel@world.std.com`

либо пишите нам:

Harvey M. Deitel (author)
Paul J. Deitel (author)
c/o Computer Science Editor
College Book Editorial
Prentice Hall
Englewood Cliffs, New Jersey 07632
Мы немедленно вам ответим.

Харви М. Дейтел
Пол Дж. Дейтел

Принципы машинной обработки данных



Цели

- Понять основные принципы организации компьютеров.
- Познакомиться с различными типами языков программирования.
- Познакомиться с историей языка С.
- Получить представление о стандартной библиотеке С.
- Понять принципы среды разработки программ на С.
- Оценить возможности С как языка для начального обучения программированию.
- Оценить преимущества С как базы для дальнейшего изучения программирования вообще и языка С++ в частности.

Содержание

1.1. Введение
1.2. Что такое компьютер?
1.3. Внутренняя организация компьютера.
1.4. Пакетная обработка, мультипрограммирование и разделение времени
1.5. Персональные вычисления, распределенные вычисления и вычисления в модели клиент/сервер
1.6. Машинные языки, языки ассемблера и языки высокого уровня.
1.7. История языка C
1.8. Стандартная библиотека C
1.9. Другие языки высокого уровня
1.10. Структурное программирование
1.11. Основные принципы среды C
1.12. Общие замечания о C и об этой книге
1.13. Concurrent C (Параллельный C)
1.14. Объектно-ориентированное программирование и C++

Резюме • Хороший стиль программирования • Советы по переносимости программ • Советы по повышению эффективности • Упражнения для самоконтроля • Ответы к упражнениям для самоконтроля • Упражнения • Рекомендуемая литература

1.1. Введение

Итак, уважаемый читатель, добро пожаловать в мир программирования на языке C! Авторы потратили на эту книгу немало труда и надеются, что чтение ее окажется для вас не только полезным, но и занимательным. C является трудным языком, который обычно изучают только опытные программисты. Поэтому данная книга в ряду учебников по C является уникальной:

- Она подойдет техническим специалистам, которые не имеют вообще, либо имеют небольшой опыт программирования.
- Она пригодится и опытным программистам, которым требуется глубокое и строгое изложение языка.

Возникает вполне естественный вопрос: каким образом одна книга способна удовлетворить потребности двух различных групп читателей? Ответ заключается в том, что в книге сделан упор на достижение ясности написания программы посредством апробированного метода «структурного программирования». Непрограммисты получают возможность изучать программирование «правильным» образом, т.е. начиная с азов. Мы попытались написать книгу в простой и достаточно ясной манере со множеством иллюстраций. И что, возможно, наиболее важно, книга снабжена большим количеством работающих С-программ с образцами выдачи при запуске их на компьютере. Первые четыре главы вводят фундаментальные принципы процесса вычислений, компьютерного программирования и языка С. Введение в программирование представлено в аспекте структурного подхода. Новички, пользовавшиеся нашим курсом, высказывают мнение, что материал этих глав представляет собой солидную фундаментальную базу для углубленного изучения С в последующих главах с 5 по 14. Опытные программисты обычно быстро прочитывают первые четыре главы и находят, что изложение материала в главах с 5 по 14 достаточно строгое и в то же время интересное. Они особенно отмечают детальное описание в этих главах указателей, строк, файлов и структур.

Многие опытные программисты оценили наше изложение материала по структурному программированию. В основном они программировали на структурном языке типа Pascal, а поскольку они никогда формально не изучали структурное программирование, то соответственно и написанный ими код был не самым лучшим. После изучения С по нашей книге они получили возможность улучшить свой стиль программирования. Поэтому является ли читатель новичком, либо опытным программистом, — здесь он найдет много полезной и интересной информации.

Большинство людей так или иначе знакомы с теми удивительными вещами, которые могут делать компьютеры. В данном курсе читатель узнает, как подавать компьютеру команды для их выполнения. Компьютерами, т.е. *аппаратной частью* (hardware) управляет *программное обеспечение* (software), а одним из наиболее популярных на сегодняшний день языков для разработки программного обеспечения является С. Данная книга описывает стандартизованную версию языка С, принятую в качестве стандарта в 1989 году Американским институтом национальных стандартов (American National Standards Instutute — ANSI) и Международной организацией стандартов (International Standards Organization — ISO).

В настоящее время почти во всех отраслях человеческой деятельности использование компьютеров все более расширяется. На фоне постоянно растущих цен стоимость вычислительной техники драматически падает благодаря значительным успехам как в разработке аппаратной части, так и в развитии программного обеспечения. Компьютеры, которые 25 лет назад занимали несколько комнат и стоили миллионы долларов, сегодня помещаются в нескольких кремниевых микросхемах стоимостью по несколько долларов каждая. По иронии судьбы кремний является наиболее распространенным материалом на Земле — это ингредиент обычного песка. Технология производства кремниевых микросхем сделала компьютеризацию настолько экономически выгодной, что в настоящее время в мире насчитывается приблизительно 150 миллионов компьютеров общего назначения, которые помогают людям в бизнесе, промышленности, политике и в повседневной жизни. Через несколько лет это число вполне может быть удвоено.

Можно ли изучать C как первый язык программирования, пользуясь, например, данной книгой? Мы полагаем, что да. Два года назад, когда в качестве первого языка люди изучали Pascal, мы решили попробовать. Была написана книга *«Как программировать на C»*, первое издание текста, который вы сейчас читаете. Сотни университетов по всему миру использовали эту книгу. Было доказано, что учебные курсы на ее основе одинаково эффективны с предшествующими курсами, ориентированными на Паскаль. Не было замечено значительной разницы за исключением того, что у студентов появилась большая мотивация для изучения языка, потому что они знают, что вероятнее всего в своей профессиональной деятельности им придется программировать на C, а не на Паскале. Студенты, изучающие C, также великолепно понимают, что будут более подготовлены для последующего изучения C++. C++ является «надмножеством» языка C, предназначенным для программистов, пишущих объектно-ориентированные программы. Более подробно о C++ мы расскажем в разделе 1.14.

Действительно, сегодня C++ привлекает настолько большое внимание, что в главе 15 (и далее) мы даем подробное введение в C++ и объектно-ориентированное программирование. Интересным явлением на рынке языков программирования является тот факт, что ведущие производители чаще поставляют объединенный продукт C/C++, нежели эти языки по отдельности. Это дает возможность пользователям продолжать программирование на C, а затем, при необходимости, постепенно перейти на C++.

Итак, перед вами книга по программированию на C. При желании читатель может связаться с нами по электронной почте через Internet по адресу deitel@world.std.com. Мы, в свою очередь, сделаем все возможное, чтобы ответить на каждое сообщение.

1.2. Что такое компьютер?

Компьютер — это устройство, способное производить вычисления, а также выполнять действия по принятию логических решений со скоростью в миллионы или даже миллиарды раз быстрее, чем человек. Например, большинство современных персональных компьютеров могут выполнять десятки миллионов арифметических операций в секунду. Человеку с калькулятором в руках потребуется несколько месяцев, чтобы выполнить такой же объем работы, какой персональный компьютер производит за одну секунду. (Вопрос: как вы узнаете, правильно ли человек складывает цифры? Как вы узнаете, правильно ли компьютер складывает цифры?) Сегодняшние самые быстрые *суперкомпьютеры* могут выполнять сотни миллиардов сложений в секунду — то есть такое количество арифметических операций под силу выполнить лишь сотням тысяч человек в течение года! А в исследовательских лабораториях уже работают компьютеры с быстродействием триллион операций в секунду.

Компьютеры обрабатывают *данные*, используя наборы инструкций, называемые *компьютерными программами*, которые заставляют компьютер выполнять определенные действия в заранее установленном порядке. Действия определяются людьми, которых называют *программистами*.

Различные устройства (такие как клавиатура, экран, диски, память и микропроцессор), составляющие компьютерную систему, называются *аппаратной частью*. Программы, выполняющиеся на компьютере, называются *программным обеспечением*. Стоимость аппаратуры в последние годы значите-

льно упала, благодаря чему компьютеры стали доступным предметом потребления. К сожалению, стоимость программного обеспечения постоянно растет, поскольку программисты разрабатывают все более мощные и сложные прикладные программы. При этом в самой технологии разработки программного обеспечения не наблюдается адекватных улучшений. В данной книге рассматриваются методы разработки программ, которые могут существенно сократить время и затраты на создание современных программных продуктов высокого качества. Эти методы включают в себя *структурное программирование*, *пошаговое нисходящее уточнение*, *разбиение на функции* и описанное в последнем разделе *объектно-ориентированное программирование*.

1.3. Внутренняя организация компьютера

Фактически каждый компьютер, вне зависимости от того, как он выглядит, можно представить в виде совокупности шести *логических блоков*. Это:

1. *Входной блок*. Это «принимающая» часть компьютера. Он получает информацию (данные и компьютерные программы) от различных *устройств ввода* и помещает ее для хранения в другие блоки, так что информация может быть в дальнейшем передана на обработку. Часто основная масса информации поступает в современные компьютеры через клавиатуру, напоминающую пишущую машинку.

2. *Блок вывода*. Он воспринимает обработанную компьютером информацию и помещает ее на различные *устройства вывода* так, чтобы сделать ее доступной для использования вне компьютера. На сегодняшний день информация обычно выводится на экран либо распечатывается на бумаге.

3. *Блок памяти*. Это хранилище информации, характеризуемое высокой скоростью доступа и относительно низкой емкостью. Блок памяти хранит информацию, полученную от блока ввода, и при необходимости немедленно предоставляет ее для обработки. Здесь также содержится информация, полученная в результате вычислений, до того момента, пока она не будет направлена блоком вывода на устройства вывода. Блок памяти называют просто *памятью*, или *первичной памятью*.

4. *Арифметико-логическое устройство (АЛУ)*. Это «перерабатывающая» часть компьютера. Она несет ответственность за выполнение арифметических операций, таких как сложение, вычитание, умножение и деление. В ней также содержатся механизмы принятия решений, которые позволяют компьютеру, например, сравнить два числа из блока памяти и определить, равны они или нет.

5. *Центральное процессорное устройство (ЦПУ)*. Это «административная» часть компьютера, являющаяся координатором работы остальных блоков. Устройство центрального процессора направляет команду входному устройству поместить информацию в блок памяти, устройству арифметики и логики — использовать информацию для вычислений, блоку вывода — отправить информацию на конкретное устройство вывода.

6. *Блок вторичного хранения данных.* Это долговременное хранилище компьютера. Программы или данные, которые не используются активно другими блоками, обычно помещаются в устройства вторичного хранения (например, диски), до того момента, пока они не понадобятся снова, возможно, через час, день, месяц или даже год.

1.4. Пакетная обработка, мультипрограммирование и разделение времени

Раньше компьютеры могли решать в каждый момент времени только одну задачу. Подобная форма работы называется однопользовательской *пакетной обработкой*. Компьютер обрабатывает данные группами, или *пакетами*, причем в каждый момент времени выполняется только одна программа из группы. В то время пользователи представляли свои программы в вычислительные центры в виде набора перфокарт и вынуждены были ждать окончательной выдачи часами или даже в течение нескольких дней.

По мере совершенствования компьютеров становилось очевидным, что однопользовательская пакетная обработка редко использует ресурсы компьютера эффективно. Возникла идея, что несколько программ, или задач, могут *разделять* между собой ресурсы компьютера, чтобы его загрузка была более рациональной. Это называется *мультипрограммированием*. Мультипрограммирование подразумевает «одновременное» выполнение на компьютере нескольких задач; компьютер разделяет свои ресурсы между задачами, на них претендующими. Но как и прежде, пользователи представляли свои программы на перфокартах и ждали свои результаты по несколько часов.

И вот в шестидесятых годах в нескольких промышленных центрах и университетах возникла новаторская концепция *разделения времени*. Разделение времени — это специальный случай мультипрограммирования, когда пользователи получают доступ к компьютеру через устройства ввода/вывода, или терминалы. В типичной компьютерной системе разделения времени может находиться несколько сотен пользователей, решающих свои задачи одновременно. Процессор выполняет небольшую часть одной задачи, затем переключается на другую. Это происходит настолько быстро, что в течение одной секунды компьютер возвращается к одной и той же задаче несколько раз. Таким образом, *с точки зрения пользователей* их задачи решаются одновременно.

1.5. Персональные вычисления, распределенные вычисления и вычисления в модели клиент/сервер

В 1977 году благодаря компании Apple Computer родился феномен персональных вычислений. Вначале это было лишь мечтой одержимых программистов-любителей. Компьютеры становились экономически доступными настолько, чтобы *люди покупали их для решения своих персональных задач*. В 1981 году компания IBM, мировой лидер по продаже компьютеров, представила на рынке Персональный Компьютер IBM. Буквально за одну ночь персональные компьютеры приобрели законное право на свое применение в бизнесе, промышленности и государственных учреждениях.

Однако эти компьютеры были изолированы друг от друга, т.е. были компьютерами-одиночками. Люди выполняли работу на своих собственных машинах, а затем передавали друг другу диски для обмена информацией. Несмотря на то, что ранние персональные компьютеры были не настолько мощными, чтобы обеспечивать разделение времени для нескольких пользователей, эти машины могли быть объединены в компьютерные сети посредством телефонных линий или в локальные сети в пределах одной организации. Это привело к появлению феномена *распределенных вычислений*, когда работа всей организации, вместо того, чтобы быть выполненной непосредственно на центральном компьютере, распределялась по сети на рабочие места, на которых и производились все расчеты. Персональные компьютеры были достаточно мощными, чтобы удовлетворить требования индивидуальных пользователей, а также справляться с задачами по электронному обмену информацией.

Современные персональные компьютеры обладают такой же вычислительной мощностью, как и машины стоимостью в миллион долларов десятилетней давности. Наиболее мощные машины — *рабочие станции* — обеспечивают индивидуальных пользователей невиданными возможностями. Информация легко распределяется по компьютерным сетям, в которых отдельные компьютеры, называемые *серверами*, хранят общий для всех набор программ и данных, которые могут быть использованы компьютерами — клиентами, входящими в состав сети. Отсюда происходит термин *клиент/сервер*. Языки C и C++ стали теми языками программирования, на которых пишут программное обеспечение для операционных систем, компьютерных сетей и для прикладных программ модели клиент/сервер.

1.6. Машинные языки, языки ассемблера и языки высокого уровня

Программисты пишут программы на различных языках программирования, некоторые из которых непосредственно понятны компьютеру, а другие нуждаются в промежуточной стадии *трансляции*. Сотни имеющихся языков могут быть подразделены на три общих типа:

1. Машинные языки
2. Ассемблерные языки
3. Языки высокого уровня

Каждый компьютер может понимать только свой *машинный язык*, который является естественным языком конкретного компьютера. Он тесно связан с его аппаратной частью. Машинные языки в общем случае состоят из последовательностей чисел (обычно нулей и единиц), которые являются командами на выполнение одиночных элементарных операций. Машинные языки являются *машинно-зависимыми*, т.е. конкретный машинный язык может быть использован только с определенным типом компьютера. Машинные языки неудобны для восприятия человеком, что можно проиллюстрировать небольшой программой, в которой к тарифной ставке прибавляются выплаты за сверхурочную работу, а результат сохраняется в переменной общей выплаты.

```
+ 1300042774
+ 1400593419
+ 1200274027
```

По мере распространения компьютеров становилось очевидным, что программирование на машинных языках тормозит развитие компьютерной техники, является очень медленным и для большинства программистов непосильным занятием. Вместо последовательностей чисел, непосредственно понятных компьютеру, программисты для представления элементарных операций стали применять англоязычные аббревиатуры, которые и сформировали основу *языков ассемблера*. Для преобразования программ, написанных на таких языках, в машинный язык были разработаны *программы-трансляторы*, называемые *ассемблерами*. Преобразование происходило со скоростью, равной быстродействию компьютера. Нижеприведенный фрагмент программы на языке ассемблера также вычисляет основную стоимость в виде суммы тарифной ставки и сверхурочных. При этом программа становится более понятной, чем аналогичная на машинном языке:

```
LOAD    BASEPAY
ADD     OVERPAY
STORE   GROSSPAY
```

С появлением языков ассемблера использование компьютеров значительно расширилось, однако все еще требовалось написание большого количества инструкций даже для реализации решения простейших задач. Для ускорения процесса программирования были разработаны *языки высокого уровня*, в которых для выполнения сложных действий достаточно написать один оператор. Программы для преобразования последовательности операторов на языке высокого уровня в машинный язык называются *компиляторами*. В языках высокого уровня инструкции, написанные программистами, зачастую выглядят как обычный текст на английском языке с применением общепринятых математических знаков. Уже знакомое нам вычисление суммарной выплаты выглядит так:

```
grossPay = basePay + overPay
```

Совершенно очевидно, что с точки зрения программистов языки высокого уровня более предпочтительны, чем любые машинные или ассемблерные языки.

1.7. История языка C

Язык C берет свое начало от двух языков, BCPL и B. В 1967 году Мартин Ричардс разработал BCPL как язык для написания системного программного обеспечения и компиляторов. В 1970 году Кен Томпсон использовал B для создания ранних версий операционной системы UNIX на компьютере DEC PDP-7. Как в BCPL, так и в B переменные не разделялись на типы — каждое значение данных занимало одно слово в памяти и ответственность за различение, например, целых и действительных чисел целиком ложилась на плечи программиста.

Язык C был разработан (на основе B) Деннисом Ричи из Bell Laboratories и впервые был реализован в 1972 году на компьютере DEC PDP-11. Известность C получил в качестве языка операционной системы UNIX. Сегодня практически все основные операционные системы написаны на C и/или C++. По проше-

ствии двух десятилетий С имеется в наличии на большинстве компьютеров. Язык С не зависит от аппаратной части, и программы, написанные на нем, могут быть *перенесены* на многие системы. С сочетает в себе основные принципы языков BCPL и B; кроме того, в нем введена типизация переменных и некоторые другие важные моменты.

В конце 70-х годов С превратился в то, что мы называем «традиционный С». Публикация в 1978 году книги Кернигана и Ричи *«Язык программирования С»* привлекла широкое внимание к языку. Эта книга стала одной из самых удачных среди изданий по компьютерным дисциплинам, выпущенных в свет.

Применение С для разных типов компьютеров (иногда называемых *аппаратными платформами*) привело к появлению различных вариантов языка, которые, несмотря на свою схожесть, зачастую были несовместимыми. Это явилось серьезной проблемой для разработчиков программных продуктов, которым требовалось разрабатывать коды, способные работать на нескольких платформах. Становилось ясно, что нужна стандартная версия С. В 1983 году Американским комитетом национальных стандартов в области компьютеров и обработке информации (X3) был учрежден технический комитет X3J11, перед которым ставилось целью выработать «однозначное и машинно-независимое определение языка С». В 1989 году стандарт был одобрен. Документ известен как ANSI/ISO 9899:1990. Копии могут быть заказаны через Американский институт национальных стандартов, адрес которого указан в предисловии. Второе издание книги Кернигана и Ричи выпущено в 1988 году и описывает эту версию С, называемую ANSI С и широко используемую в мире (Ке88).

Совет по переносимости программ 1.1

Благодаря тому, что С является машинно-независимым и широко доступным языком, прикладные программы, написанные на С, могут работать практически без изменений на большинстве компьютерных систем.

1.8. Стандартная библиотека С

Как вы узнаете в главе 5, программы, написанные на С, состоят из модулей, или фрагментов, называемых *функциями*. Пользователь сам может написать необходимые для создания программы функции, однако большинство программистов предпочитают применять уже существующие, которые находятся в *стандартной библиотеке С*. Таким образом, есть два аспекта изучения языка. Первый — изучение непосредственно С и второй — умение пользоваться стандартной библиотекой. В данной книге рассматриваются многие функции из стандартной библиотеки, все они приводятся в приложении Б (так, как их определяет ANSI С). Книга Плогера (Р192) будет полезна программистам, желающим приобрести глубокие знания и понимание работы стандартных функций, их применения при написании компактных кодов.

В данном курсе читателю будет предложен *блочный метод* построения программы. Не следует изобретать велосипед. Применение готовых частей называется *повторным использованием кода* — это ключевой момент в развивающейся сейчас области объектно-ориентированного программирования (глава 15). В языке С при формировании программы вы будете использовать следующие «строительные блоки»:

- Функции стандартной библиотеки C
- Функции, которые вы создадите сами
- Функции, написанные другими людьми

Преимущество создания своих собственных функций состоит в том, что вы отлично знаете и понимаете, как они работают. У вас появится возможность проследить за работой кода C. Ну а недостаток — временные затраты, которые потребуются для создания новых функций.

При использовании существующих функций не приходится изобретать велосипед. В случае применения стандартных функций ANSI, которые, как вы знаете, очень тщательно написаны, ваши программы с большей вероятностью могут быть перенесены на другие типы компьютеров.

Совет по повышению эффективности 1.1

Использование функций стандартной библиотеки ANSI вместо написания собственных аналогичных функций может улучшить характеристики программы, т. к. стандартные функции написаны весьма тщательно и работают очень эффективно.

Совет по переносимости программ 1.2

Использование функций стандартной библиотеки ANSI вместо написания собственных аналогичных функций может способствовать переносимости программ, т. к. стандартные функции реализованы практически во всех существующих вариантах C.

1.9. Другие языки высокого уровня

Несмотря на то, что были разработаны сотни языков высокого уровня, только некоторые из них нашли широкое применение. FORTRAN (FORmula TRANslator) разработан фирмой IBM между 1954 и 1957 годами для использования в научных и инженерных программах, требующих сложных математических вычислений. FORTRAN широко применяется и до сих пор.

COBOL (COmmon Business Oriented Language) разработан в 1959 году группой, в которую входили производители и пользователи компьютеров. COBOL главным образом используется в коммерческих прикладных программах, когда необходима высокая точность при обработке большого количества данных. Сегодня больше половины программного обеспечения, предназначенного для использования в бизнесе, все еще написано на языке COBOL. Свыше миллиона человек в настоящее время зарабатывают на жизнь, программируя на COBOL.

Pascal был создан почти в то же время, что и C. Он предназначен для академических целей. В следующем разделе мы остановимся на языке Pascal более подробно.

1.10. Структурное программирование

В шестидесятых годах при разработке программного обеспечения программисты испытывали серьезные затруднения. Время разработки обычно затягивалось свыше установленных сроков, стоимость работ значительно превышала бюджет, а окончательный продукт был весьма ненадежен. Люди начали

понимать, что разработка программного продукта является делом более сложным, чем они себе представляли. В результате научных исследований, проведенных в шестидесятых годах, возникла концепция *структурного программирования*, т.е. такого подхода к написанию программ, при котором они становятся более ясными для понимания, более корректными и лучше приспособленными для модификации в дальнейшем. В главах 3 и 4 приводится обзор принципов структурного программирования. Ниже мы остановимся лишь на обсуждении развития структурного программирования на С.

Одним из знаменательных результатов научных исследований в области программирования явилось создание в 1971 году профессором Никласом Виртом языка программирования Pascal. Pascal, названный по имени математика и философа семнадцатого века Блеза Паскаля, был разработан для обучения структурному программированию в академических кругах и быстро распространился в университетах для изучения в качестве первого языка программирования. К сожалению, в нем отсутствуют свойства, благодаря которым его можно было бы использовать в коммерческой и индустриальной сферах деятельности и, как следствие, в этих областях он не получил широкого распространения. Однако можно отметить, что реальное значение этого языка в том, что он лег в основу языка программирования Ada. Разработка языка Ada финансировалась Министерством обороны США в конце 70-х — начале 80-х годов. Для обслуживания огромного комплекса МО использовались сотни различных языков. Для их замены было принято решение создать один язык программирования, который бы удовлетворял всем требованиям. Для этой цели был выбран Pascal, однако конечный продукт — Ada — значительно от него отличается. Новый язык программирования был назван по имени леди Ады Лавлейс, дочери поэта лорда Байрона. Считается, что Ада Лавлейс в начале 1800-х написала первую в мире компьютерную программу. Одной из основных черт языка Ada является *многозадачность*, которая позволяет нескольким процессам протекать параллельно. Остальные языки высокого уровня, которые мы обсуждали, включая С и C++, допускают выполнение в данный момент только одной задачи.

1.11. Основные принципы среды С

Все С-системы в общем состоят из трех частей: среды программирования, собственно языка и стандартной библиотеки С. В процессе подготовки С-программы проходят через шесть стадий (рис. 1.1). Это: *редактирование*, *препроцессорная обработка*, *компиляция*, *компоновка*, *загрузка* и *исполнение*. Здесь мы имеем в виду типичную UNIX-систему С. В случае если читатель использует другую операционную систему, ему следует обратиться к соответствующему справочному руководству или к более опытному коллеге за разъяснениями, как эти задачи выполняются в его среде.

Первая фаза состоит из редактирования файла при помощи *программы-редактора*, в которой программист набирает программу и производит необходимые исправления. Затем программа сохраняется на вторичном запоминающем устройстве, например, диске. Имена файлов, содержащих программы на С, должны иметь расширение **.c**. В системах UNIX широко используются два редактора — **vi** и **emacs**. Пакеты C/C++, такие, как Borland C++ для IBM и совместимых персональных компьютеров или Symantec C++ для Apple Macintosh, обладают встроенными редакторами, интегрированными в среду программирования. Мы предполагаем, что читатель знает, как отредактировать программу.

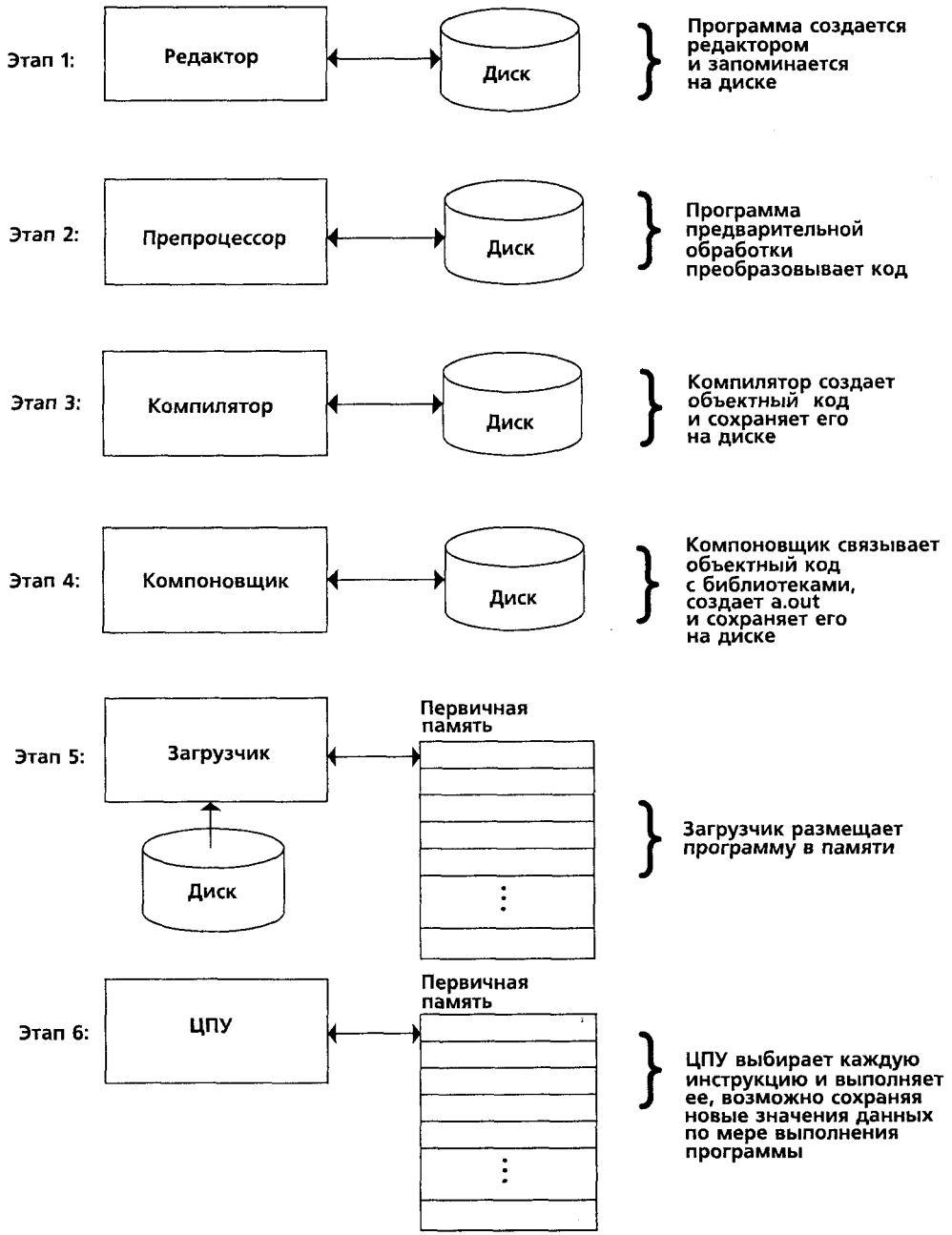


Рис. 1.1. Типичная среда C

Далее, программист дает команду на *компиляцию* программы. Компилятор транслирует C-программу в код машинного языка (который также называют *объектным кодом*). В C-системе перед началом фазы трансляции автоматически выполняется программа *препроцессора*. Препроцессор C подчиняется

специальным командам — директивам препроцессора, которые определяют, какие действия должны быть выполнены перед компиляцией программы. Обычно это добавление других файлов к компилируемому и замена специальных символов в тексте программы. В общих чертах директивы препроцессора описываются в начальных главах книги, а детальному их рассмотрению посвящена глава 13. Препроцессор автоматически запускается компилятором перед тем, как начнется преобразование программы в машинный язык.

Четвертая фаза называется *компоновкой*. Типичные С-программы содержат обращения к функциям, чьи определения находятся либо в стандартных библиотеках, либо в библиотеках, созданных группой пользователей в процессе работы над конкретным проектом. То есть объектный код, получившийся в результате работы компилятора, содержит «дыры», обусловленные отсутствием определений функций, к которым происходит обращение. Компоновщик связывает объектный код с определениями отсутствующих функций, т.е. вырабатывает *исполняемый образ* (в котором уже нет отсутствующих частей). Например, для компиляции и компоновки программы с именем **welcome.c** следует после подсказки UNIX напечатать

```
cc  welcome.c
```

и нажать клавишу Enter. В случае корректной компиляции и компоновки появится файл **a.out**, являющимся исполняемым кодом программы **welcome.c**. Пятая фаза называется *загрузкой*. Перед выполнением программа должна быть размещена в памяти. Эту задачу выполняет загрузчик, который помещает в память исполняемый код с диска.

И наконец, компьютер, под контролем ЦПУ, выполняет программу, последовательно прочитывая ее инструкции одну за другой. Для загрузки и выполнения программы в системе UNIX после приглашения мы печатаем **a.out** и нажимаем Enter.

Большинство С-программ получают и (или) выводят данные. Конкретные функции С получают входные данные от **stdin** (standard input device — стандартное устройство ввода), которое, как правило, ассоциировано с клавиатурой; однако, **stdin** может быть подключено и к другому устройству. Вывод данных осуществляется на **stdout** (standard output device — стандартное устройство вывода), которым обычно является экран компьютера, однако также может быть подключено к другому устройству. Когда мы говорим, что программа печатает результат, мы подразумеваем, что результат высвечивается на экране. Данные могут выводиться на другие устройства, такие как диск или принтер. Имеется также **stderr** (standard error device — стандартное устройство ошибок). Устройство **stderr** (обычно ассоциированное с экраном) используется для вывода сообщений об ошибках. Часто для вывода данных, т.е. для **stdout**, вместо экрана применяют другие устройства, а экран используют для вывода сообщений об ошибках, т.е. в качестве **stderr**, чтобы пользователь как можно быстрее был информирован о возникающих ошибках.

1.12. Общие замечания о С и этой книге

С — это трудный язык. Опытные программисты зачастую гордятся своим умением писать запутанные и загадочные тексты программ. Однако это просто плохой стиль программирования. В результате программа становится неудобочитаемой, увеличивается вероятность сбоев и затрудняется ее отладка и

проверка. Данная книга предназначена для начинающих программистов, поэтому мы делаем упор на написание ясных, хорошо структурированных программ. Одной из ключевых целей этой книги является достижение четкости при написании программы через применение апробированной методики структурного программирования и благодаря связанному с ней хорошему стилю программирования.

Хороший стиль программирования 1.1

Пишите программы в простой, четкой манере. Подобный стиль иногда называют KIS («keep it simple» — будьте проще).

Возможно, читатель слышал, что C — переносимый язык и что написанные на нем программы могут работать на разных компьютерах. Однако переносимость является весьма труднодостижимым свойством. Стандартный документ ANSI насчитывает 11 страниц, посвященным проблеме переносимости. На тему переносимости C написаны целые книги (Ja89), (Ra90).

Совет по переносимости программ 1.3

Несмотря на возможность написания переносимых программ существует много проблем совместимости между различными версиями C и разными компьютерными системами, что затрудняет достижение переносимости. Само по себе написание программ на C не обеспечивает их переносимости.

Мы очень тщательно проработали стандартный документ ANSI C и в соответствии с ним подготовили нашу книгу. Однако C — очень богатый язык и некоторые его тонкости в книге не рассматриваются. Если читателю понадобится дополнительное техническое описание ANSI C, то мы рекомендуем ему самому прочитать стандарт ANSI C или книгу Кернигана и Ричи (Ke88). Мы ограничиваем наше обсуждение ANSI C, который по многим свойствам несовместим с ранними версиями языка, и в этой книге читатель может найти некоторые программы, не работающие со старыми C-компиляторами.

Хороший стиль программирования 1.2

Читайте руководства пользователя по версиям C, с которыми вы работаете. Частое обращение к ним позволит вам узнать много полезного об особенностях языка C и поможет их корректно использовать.

Хороший стиль программирования 1.3

Хорошими учителями являются компьютер и компилятор. Если вы не уверены в том, как работает та или иная конструкция C, то следует написать пробную программу, откомпилировать, запустить ее и посмотреть, что получится.

1.13. Concurrent C (Параллельный C)

В результате продолжающейся исследовательской работы в Bell Laboratories были разработаны другие версии C. Гехани (Ge89) разработал *Concurrent C* (Параллельный C) — надмножество, обладающее возможностями для парал-

лельного запуска нескольких задач. Языки типа Concurrent C, а также операционные системы, поддерживающие параллельную работу, станут более популярны в следующем десятилетии с ростом использования *мультипроцессоров* (т.е. компьютеров с несколькими ЦПУ). В настоящее время Concurrent C не более чем исследовательский язык. Книги по операционным системам (De90), как правило, уделяют значительное внимание параллельному программированию.

1.14. Объектно-ориентированное программирование и C++

В лаборатории Bell Бьерном Страуструпом (St86) было разработано еще одно надмножество C, а именно C++. Язык C++ имеет целый ряд качеств, «украшающих» обычный C. Но самое главное, он предоставляет возможности для *объектно-ориентированного программирования*.

Объекты — это в принципе многократно используемые компоненты программного обеспечения, которые моделируют реальные сущности. В мире программных продуктов произошла революция. Быстрое, корректное и экономически выгодное построение программного обеспечения все еще остается труднодостижимой целью, а время таково, что воздух пронизан потребностью в новом, мощном программном обеспечении.

В настоящее время разработчики сознают, что применение объектно-ориентированного подхода способно повысить производительность рабочих коллективов в 10-100 раз по сравнению со стандартной методикой программирования.

Разработано много объектно-ориентированных языков. Однако существует убежденность, что именно C++ станет доминантным системным языком середины — конца 90-х годов.

Большинство людей склоняется к тому, что сегодня наилучшей стратегией обучения является первоначальное изучение C, а затем C++. Поэтому главы с 15 по 21 посвящены введению в объектно-ориентированное программирование и C++. Мы надеемся, что они будут оценены читателем, и что после прочтения книги он продолжит изучение C++.

Резюме

- Аппаратной частью компьютера (hardware) управляет программное обеспечение (software).
- ANSI-C — это версия языка программирования C, принятая в качестве стандарта в 1989 году Американским институтом национальных стандартов (ANSI) и Международной организацией стандартов (ISO).
- Компьютеры, которые 25 лет назад занимали несколько комнат и стоили миллионы долларов, сегодня помещаются в нескольких кремниевых микросхем стоимостью по несколько долларов каждая.
- В настоящее время в мире насчитывается приблизительно 150 миллионов компьютеров общего назначения, которые помогают людям в бизнесе, промышленности, политике и в повседневной жизни. Через несколько лет это число может быть удвоено.

- Компьютер — это устройство, способное производить вычисления, а также выполнять действия по принятию логических решений со скоростью в миллионы или даже миллиарды раз быстрее, чем человек.
- Компьютеры обрабатывают данные под управлением компьютерных программ.
- Различные устройства (такие, как клавиатура, экран, диски, память и процессорные устройства), составляющие компьютерную систему, называются ее аппаратной частью.
- Компьютерные программы, выполняющиеся в компьютере, называются программным обеспечением.
- Входной блок — это «принимающая» часть компьютера. Большинство информации в современные компьютеры поступает через клавиатуру.
- Блок вывода — это часть компьютера, отвечающая за выдачу информации. На сегодняшний день большинство информации выводится на экран либо распечатывается на бумаге.
- Блок памяти — это «хранилище» компьютера, именуемое памятью, или первичной памятью.
- Арифметико-логическое устройство (АЛУ) выполняет вычисления и ответственно за процесс принятия решений.
- Центральное процессорное устройство (ЦПУ) — это «административная» часть компьютера, являющаяся координатором работы остальных блоков.
- Программы или данные, которые не используются активно другими блоками, обычно помещаются на устройства вторичного хранения (например, диски), пока они не понадобятся снова.
- При однозадачной пакетной обработке данные обрабатываются группами, или пакетами, причем компьютер выполняет в каждый момент времени одну-единственную программу.
- Мультипрограммирование подразумевает одновременное выполнение на компьютере нескольких задач — компьютер распределяет свои ресурсы между ними.
- Разделение времени — это специальный случай мультипрограммирования, когда пользователи получают доступ к компьютеру через устройства ввода/вывода, или терминалы. С точки зрения пользователей их задачи решаются одновременно.
- При распределенных вычислениях работа всей организации распределяется по сети на рабочие места, на которых и производятся все расчеты.
- Серверы хранят общий для всех пользователей набор программ и данных, которые могут быть использованы компьютерами — клиентами, входящими в состав сети. Отсюда происходит термин «модель клиент/сервер».
- Каждый компьютер может непосредственно понимать только свой машинный язык.

- Машинные языки в общем случае состоят из последовательностей чисел (в конечном счете сводящихся к нулям и единицам), которые являются командами на выполнение одиночных элементарных операций. Машинные языки являются машинно-зависимыми.
- Основу языков ассемблера образуют англоязычные аббревиатуры. Ассемблеры транслируют программы, написанные на ассемблерных языках, в машинный язык.
- Программы для преобразования последовательности операторов на языке высокого уровня в машинный язык называются компиляторами.
- C известен как язык разработки операционной системы UNIX.
- Программы, написанные на C, могут быть перенесены на большинство существующих компьютерных систем.
- В 1989 году был одобрен стандарт ANSI C.
- FORTRAN (FORmula TRANslator) используется в научных и инженерных программах.
- COBOL (COmmon Business Oriented Language) главным образом используется в коммерческих прикладных программах, когда необходима высокая точность при обработке большого количества данных.
- Структурное программирование — это такой подход к написанию программ, при котором они становятся более ясными для понимания, более корректными и лучше подходящими для дальнейших модификаций.
- Pascal был разработан для обучения структурному программированию в академических кругах.
- Разработка языка Ada финансировалась Министерством обороны США. В качестве основы для его создания был выбран Pascal.
- Одной из основных черт языка Ada является многозадачность, которая позволяет нескольким процессам протекать параллельно.
- Все C-системы в общем состоят из трех частей: среды, языка программирования и стандартной библиотеки C. Библиотечные функции не являются собственно частью языка C; они выполняют такие операции, как ввод/вывод данных, а также математические вычисления.
- В процессе подготовки C-программы проходят через шесть стадий: редактирование, препроцессорная обработка, компиляция, компоновка, загрузка и выполнение.
- Программист набирает программу и производит необходимые исправления при помощи программы-редактора.
- Компилятор транслирует программу на C в код машинного языка (который также называют объектным кодом).
- Препроцессор C подчиняется специальным командам — директивам препроцессора, которые обычно указывают, что к компилируемому файлу следует добавить другие файлы и должна быть произведена замена специальных символов в тексте программы.

- Компоновщик связывает объектный код с определениями отсутствующих функций, т.е. вырабатывает исполняемый код (в котором уже нет отсутствующих частей).
- Загрузчик помещает в память исполняемый код с диска.
- Компьютер, под контролем ЦПУ, выполняет программу, последовательно прочитывая ее инструкции одну за другой.
- Некоторые функции С (например, **scanf**) получают входные данные от **stdin** (стандартного устройства ввода), которое, как правило, ассоциировано с клавиатурой.
- Вывод данных осуществляется на **stdout** (стандартное устройство вывода), которым обычно является экран компьютера.
- Имеется также **stderr** (стандартное устройство ошибок). Устройство **stderr** (обычно ассоциированное с экраном) используется для вывода сообщений об ошибках.
- Несмотря на возможность написания переносимых программ, существует много проблем совместимости между различными версиями С и разными компьютерными системами, что затрудняет достижение переносимости.
- Concurrent C — это надмножество С, обладающее возможностями для параллельного запуска нескольких задач.
- C++ предоставляет возможности для объектно-ориентированного программирования.
- Объекты — это в принципе многократно используемые компоненты программного обеспечения, которые моделируют реальные сущности.
- Существует общепринятое мнение, что именно C++ станет ведущим системным языком середины — конца 90-х годов.

Терминология

C
C++
COBOL
Concurrent C
CPU
FORTRAN
Pascal
UNIX
аппаратная платформа
аппаратная часть
блок памяти
блочный метод построения программ
ввод/вывод (I/O)
входной блок
выходной блок
данные
естественный язык компьютера
загрузчик
задача
запуск программы
исполнение программы
исполняемый образ
клиент
компилятор
компоновщик
компьютер
компьютерная программа
логические блоки
машинно-зависимый
машинно-независимый
машинный язык
многозадачность
модель клиент/сервер
мультипрограммирование
мультипроцессор
нисходящее пошаговое уточнение
объект
объектно-ориентированное программирование

объектный код
окружение (рабочая среда)
память
первичная память
переносимость
персональный компьютер
повторное использование программного кода
препроцессор С
программа-транслятор
программист
программное обеспечение
рабочая станция
разбиение на функции
разделение времени
распределенные вычисления
редактор
сохранение программы
стандартная библиотека С
стандартная ошибка (**stderr**)
стандартный ввод (**stdin**)
стандартный вывод (**stdout**)
структурное программирование
суперкомпьютер
терминал
устройство ввода
устройство вывода
файловый сервер
функция
центральное процессорное устройство (CPU)
экран
язык высокого уровня
язык программирования
ясность

Хороший стиль программирования

1.1. Пишите программы в простой, четкой манере. Подобный стиль иногда называют KIS («keep it simple» — будьте проще).

1.2. Читайте руководства пользователя по версиям С, с которыми вы работаете. Частое обращение к ним позволит вам узнать много полезного об особенностях языка С и поможет их корректно использовать.

1.3. Хорошими учителями являются компьютер и компилятор. Если вы не уверены в том, как работает та или иная конструкция С, то следует написать пробную программу, откомпилировать, запустить ее и посмотреть, что получится.

Советы по переносимости программ

1.1. Благодаря тому, что С является машинно-независимым и широко доступным языком, прикладные программы, написанные на С, могут работать практически без изменений на большинстве компьютерных систем.

1.2. Использование функций стандартной библиотеки ANSI вместо написания собственных аналогичных функций может улучшить переносимость программы, т.к. стандартные функции реализованы фактически во всех существующих вариантах С.

1.3. Несмотря на возможность написания переносимых программ, существует много проблем совместимости между различными версиями С и разными компьютерными системами, что затрудняет достижение переносимости. Само по себе написание программ на С не обеспечивает переносимости.

Советы по повышению эффективности

1.1. Использование функций стандартной библиотеки ANSI вместо ни-сания собственных аналогичных функций может улучшить характеристики программы, т.к. стандартные функции написаны весьма тщательно и работают очень эффективно.

Упражнения для самоконтроля

1.1. Заполните пробелы в каждом из следующих предложений:

а) Компания, которая ввела в мир феномен персональных вычислений, называется _____.

б) Компьютер, благодаря которому персональные вычисления были введены в практику промышленности и бизнеса, называется _____.

в) Компьютеры обрабатывают данные, следуя наборам инструкций, называемым компьютерными _____.

г) Шестью ключевыми логическими блоками компьютера являются: _____, _____, _____, _____, _____ и _____.

д) _____ — это специальный случай мультипрограммирования, при котором пользователи получают доступ к компьютеру посредством устройств под названием «терминалы».

е) В главе обсуждаются три класса языков: _____, _____ и _____.

ж) Программы, которые транслируют код с языков высокого уровня на машинный язык, называются _____.

з) С широко известен как язык, на котором написана операционная система _____.

и) Книга рассматривает версию С, которая называется _____ С и была стандартизирована Американским национальным институтом стандартов.

к) Язык _____ был разработан Виртом для обучения структурному программированию в университетах.

л) В Министерстве обороны был разработан язык Ada, который характеризуется _____, позволяющей программистам запускать несколько задач параллельно.

1.2. Заполните пробелы в следующих предложениях о среде С.

а) С-программы обычно набирают на компьютере, используя программу _____.

б) В С-системе перед началом фазы трансляции автоматически выполняется программа _____.

в) Двумя наиболее частыми операциями во время препроцессорной обработки являются _____ и _____.

г) Программа _____ для формирования исполняемого кода объединяет выходные данные компилятора с различными библиотечными функциями.

д) Программа _____ перемещает исполняемый код с диска в память.

е) Для того, чтобы загрузить и выполнить только что откомпилированную программу в системе UNIX, следует ввести _____.

Ответы к упражнениям для самоконтроля

1.1. а) Apple, б) IBM PC, в) программами, г) блок ввода, блок вывода, блок памяти, арифметико-логическое устройство (АЛУ), центральное процессорное устройство, блок вторичного хранения данных, д) разделение времени, е) машинные языки, языки ассемблера и языки высокого уровня, ж) компиляторами, з) UNIX, и) ANSI, к) Pascal, л) многозадачностью.

1.2. а) редактора, б) препроцессора, в) включение в компилируемый файл других файлов, замена специальных символов в тексте программы, г) компоновщика, д) загрузчика, е) **a.out**.

Упражнения

1.3. Ответьте, к какой группе — аппаратной части или программному обеспечению — относятся следующие объекты:

а) ЦПУ

б) Компилятор C

в) АЛУ

г) Препроцессор C

д) Блок ввода

е) Программа текстового редактора

1.4. Для чего вам может понадобиться писать программу на машинно-независимом языке вместо того, чтобы писать ее на машинно-зависимом? Почему машинно-зависимый язык может оказаться более подходящим для написания определенных типов программ?

1.5. Программы-трансляторы, такие как ассемблеры и компиляторы, преобразуют программы с одного языка (называемого *исходным*) на другой (называемый *объектным* языком). Определите, какие из следующих утверждений справедливы, а какие нет:

а) Компилятор транслирует программы, написанные на языке высокого уровня, в объектный язык.

б) Ассемблер транслирует программы с исходного языка на машинный язык.

в) Компилятор преобразует программы на исходном языке в программы на объектном языке.

г) Языки высокого уровня, как правило, машинно-зависимы.

д) Перед запуском программы на компьютере программа, написанная на машинном языке, нуждается в трансляции.

1.6. Заполните пробелы в каждом из следующих предложений:

а) Устройства, посредством которых пользователи получают доступ к системам с разделением времени, называются _____.

б) Компьютерная программа, которая преобразует коды ассемблерных программ в программы на машинном языке, называется _____.

в) Логический блок компьютера, который получает информацию извне для использования внутри компьютера, называется _____.

г) Процесс написания инструкций компьютеру для решения конкретных задач называется _____.

д) Язык какого типа использует англоязычные аббревиатуры для команд на машинном языке? _____.

е) Какие имеются шесть логических блоков компьютера? _____.

ж) Какой логический блок компьютера посылает уже обработанную информацию на различные устройства, так что информация может быть использована вне компьютера? _____.

з) Общим названием программ, которые преобразуют программы, написанные на конкретном компьютерном языке, в машинный язык, является _____.

и) Какой логический блок компьютера хранит информацию? _____.

к) Какой логический блок компьютера выполняет вычисления? _____.

л) Какой логический блок компьютера принимает логические решения? _____.

м) Какая общепринятая аббревиатура используется для обозначения управляющего блока компьютера? _____.

н) Какой уровень компьютерного языка наиболее удобен программисту для быстрого и легкого написания программ? _____.

о) Общепризнанный ориентированный на бизнес современный язык программирования — это _____.

п) Единственный язык, непосредственно понятный компьютеру, называется _____ компьютера.

р) Какой логический блок компьютера координирует деятельность всех остальных блоков? _____.

1.7. Определите, какие из следующих утверждений являются верными, а какие нет. Объясните свои ответы:

а) Машинные языки, как правило, машинно-зависимы.

б) При разделении времени несколько пользователей действительно одновременно используют компьютер.

в) Как и другие языки высокого уровня, C рассматривается как машинно-независимый язык.

1.8. Обсудите значение каждого из следующих имен в среде UNIX:

а) **stdin**

б) **stdout**

в) **stderr**

1.9. Какой ключевой особенностью обладает Concurrent C, которой нет в ANSI C?

1.10. Почему сегодня уделяется большое внимание объектно-ориентированному программированию вообще и языку C++ в частности?

Рекомендуемая литература

(An90) ANSI, American National Standard for Information Systems-Programming Language C (ANSI Document ANSI/ISO 9899: 1990), New York. NY: American National Standards Institute. 1990.

Это первоисточник по ANSI C. Распространение документа осуществляется Американским институтом национальных стандартов, 1430 Broadway, New York 10018.

(De90) Deitel, H. M., Operating Systems (Second Edition), Reading. MA: Addison-Wesley Publishing Company, 1990.

Книга о традиционном программировании операционных систем. В главах 4 и 5 широко обсуждается параллельное программирование.

(Ge89) Gehani, N., and W. D. Roome, The Concurrent C Programming Language. Summit, NJ: Silicon Press. 1989.

Первоисточник по Concurrent C — надмножества C, которое обеспечивает возможность организации параллельной работы нескольких задач.

(Ja89) Jaeschke. R., Portability and the C Language, Indianapolis, IN: Hayden Books, 1989.

Обсуждается написание переносимых программ на языке C. Джэшке принимал участие в работе комитетов по ANSI C и по ISO C.

(Ke88) Kernighan, B. W., and D. M. Ritchie, The C Programming Language (Second Edition). Englewood Cliffs. NJ: Prentice Hall. 1988.

Книга, ставшая в своем роде классической. Интенсивно использовалась на курсах языка C и на семинарах по подготовке программистов, содержит богатый справочный материал. Ричи является автором языка C и одним из создателей операционной системы UNIX.

(P192) Plauger, P. J., The Standard C Library. Englewood Cliffs. NJ: Prentice Hall. 1992.

Приводятся примеры с использованием функций стандартной библиотеки C. Плогер работал начальником подкомитета при комитете, который разрабатывал стандарт ANSI C. Сейчас является главой комитета ISO, занимающегося проблемами C.

(Ra90) Rabinowitz, H., and C. Schaap. Portable C, Englewood Cliffs. NJ: Prentice Hall. 1990.

Книга написана в качестве учебника по курсу о проблемах переносимости, читаемого в AT&T Bell Laboratories. Рабинович работает в лаборатории разработки искусственного интеллекта в корпорации NYEEX, а Шаап — глава корпорации Delft Consulting.

(Ri78) Ritchie. D. M.; S. C. Johnson; M. E. Lesk; and B. W. Kernighan, «UNIX Time-Sharing System: The C Programming Language," The Bell System Technical Journal, Vol. 57, No. 6. Part 2. July-August 1978. pp. 1991–2019.

Одна из классических статей, описывающих язык С. Впервые опубликована в специальном выпуске Bell System Technical Journal, посвященном операционной системе UNIX: «UNIX Time-Sharing System».

(Ri84) Ritchie. D. M., «The UNIX System: The Evolution of the UNIX Time-Sharing System,» AT&T Bell Laboratories Technical Journal, Vol. 63, No. 8, Part 2, October 1984. pp. 1577–1593.

Классическая статья по операционной системе UNIX. Впервые опубликована в специальном издании Bell System Technical Journal, полностью посвященном операционной системе UNIX: «The UNIX System.»

(Ro84) Rosler, L., «The UNIX System: The Evolution of C — Past and Future.» AT&T Bell Laboratories Technical Journal, Vol: 63, No. 8, Part 2, October 1984. pp. 1685–1699.

Отлично дополняет (Ri78) и будет полезна читателям, интересующимся историей языка С и проблемами его стандартизации. Впервые опубликована в специальном издании Bell System Technical Journal, полностью посвященном операционной системе UNIX: «The UNIX System.»

(St84) Stroustrup, B., «The UNIX System: Data Abstraction in C,» AT&T Bell Laboratories Technical Journal, Vol. 63. No. 8. Part 2, October 1984. pp. 1701–1732.

Классическая статья по C++. Впервые опубликована в специальном издании Bell System Technical Journal, полностью посвященном операционной системе UNIX: «The UNIX System.»

(St91) Stroustrup, B. The C++ Programming Language (Second Edition), Reading, MA: Addison-Wesley Series in Computer Science, 1991.

Первоисточник по C++, надмножества С, включающего в себя специальные возможности для объектно-ориентированного программирования. Страуструп разработал C++ в лаборатории AT&T Bell Laboratories.

(To89) Tondo, C. L., and S. E. Gimpel, The C Answer Book, Englewood Cliffs, NJ: Prentice Hall.1989.

Книга содержит ответы к упражнениям, приведенным в книге Кернигана и Ричи (Ке88). Авторы демонстрируют образцовый стиль программирования, а также обсуждают проблемы выбора схемы написания программы.

Введение в программирование на C



Цели

- Научиться писать простейшие программы на C.
- Научиться писать простые операторы ввода и вывода.
- Познакомиться с базовыми типами данных.
- Понять принципы организации компьютерной памяти.
- Научиться использованию арифметических операций.
- Изучить приоритеты арифметических операций.
- Научиться писать простейшие операторы принятия решений.

Содержание

2.1. Введение
2.2. Простая программа на С: печать строки текста
2.3. Еще одна простая программа на С: сложение двух целых чисел
2.4. Общие понятия о памяти компьютера
2.5. Арифметика в С
2.6. Принятие решений: операции равенства и отношения

Резюме • Распространенные ошибки программирования • Хороший стиль программирования • Советы по переносимости программ • Упражнения для самоконтроля • Ответы на упражнения для самоконтроля • Упражнения

2.1. Введение

Язык С способствует структурному и дисциплинированному подходу к написанию компьютерных программ. В этой главе даны вводные замечания, касающиеся программирования на С, и представлено несколько примеров, иллюстрирующих многие важные особенности языка. Каждый пример посвящен тщательному анализу какого-либо отдельного оператора. Главы 3 и 4 представляют собой введение в *структурное программирование* на С. Далее структурный подход будет использоваться до конца книги.

2.2. Простая программа на С: печать строки текста

Язык С использует нотацию, которая может показаться странной людям, никогда не программировавшим компьютеры. Мы начнем с рассмотрения довольно простой программы на С. Этот пример — печать строки текста. Программа и ее результат, выведенный на экран, представлены на рис. 2.1.

```
/* Первая программа на С */

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

```
Welcome to C!
```

Рис. 2.1. Текст программы печати

Несмотря на простоту программы, с ее помощью можно проиллюстрировать некоторые важные особенности языка C. Рассмотрим подробно каждую строчку программы. Строка

```
/* Первая программа на C */
```

начинается символами /* и заканчивается символами */, означающими, что эта строка является *комментарием*. Программисты вставляют в код комментарии для документирования программ и для того, чтобы сделать их более удобочитаемыми. Комментарии не оказывают никакого влияния на работу компьютера во время исполнения программы. Компилятор C просто игнорирует комментарии, не создавая для них никакого машинного объектного кода. Комментарий **Первая программа на C** просто объясняет назначение программы. Комментарии также помогают другим людям прочитать и понять вашу программу, однако слишком многословные комментарии могут, наоборот, затруднить ее прочтение.

Распространенная ошибка программирования 2.1

Часто забывают закончить комментарий символами */.

Распространенная ошибка программирования 2.2

Часто начинают комментарий символами */, а заканчивают символами /*.

Строка

```
main()
```

должна обязательно присутствовать в каждой программе. Скобки после **main** означают, что **main** является «строительным блоком» программы, называемым *функцией*. Программа C может содержать одну или более количество функций, однако одна из функций обязательно должна быть **main**.

Хороший стиль программирования 2.1

Каждая функция должна быть предварена комментарием, объясняющим ее назначение.

Левая фигурная скобка ({) должна предварять *тело* каждой функции. Соответственно *правая фигурная скобка* должна стоять в конце каждой функции. Эта пара скобок и часть программы между ними называется *блоком*. Блок — важная программная единица в C.

Строка

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

дает компьютеру команду выполнить *действие*, а именно вывести на экран *строку* символов, находящуюся внутри кавычек. Такую строку называют *символьной строкой*, *сообщением* или *литералом*. Вся строка, включая **printf**, *аргументы* внутри круглых скобок и *точку с запятой* (;), называется *оператором*. Каждый оператор должен заканчиваться точкой с запятой (иногда называемой *символом конца оператора*). Результатом выполнения оператора **printf** является вывод сообщения **Welcome to C!** на экран. Символы обычно печатаются именно так, как они записаны внутри двойных кавычек в операторе **printf**. Заметьте, что символы **\n** не появились на экране. Обратная косая черта (\) называется *esc-символом*. Он указывает, что **printf** предстоит

выполнить нечто нестандартное. Когда встречается обратная косая черта, **printf** считывает следующий за ним символ и, объединяя его с обратной косой четой, создает *esc-код*. Esc-код **\n** означает *новую строку*, результатом является перевод курсора на начало следующей строки на экране. Некоторые другие esc-коды представлены на рис. 2.2. Функция **printf** — одна из многих функций, входящих в *стандартную библиотеку* C (список функций стандартной библиотеки дан в Приложении Б).

Esc-код	Описание
\n	Новая строка. Перемещает курсор в начало новой строки.
\t	Горизонтальная табуляция. Перемещает курсор в следующую позицию табуляции.
\r	Возврат каретки. Перемещение курсора в начало текущей строки без перемещения на следующую строку.
\a	Звуковой системный сигнал оповещения.
\\	Обратная косая черта. Вывод на экран символа \ при помощи оператора **printf**
\"	Двойные кавычки. Вывод на экран символа двойных кавычек посредством оператора **printf**

Рис. 2.2. Некоторые распространенные esc-коды

Два последних esc-кода на рис. 2.2. могут показаться странными. Дело в том, что обратная косая черта воспринимается **printf** не так, как все остальные символы; **printf** распознает его как esc-символ, а не как символ, который необходимо отобразить, поэтому используется двойная черта (\\), чтобы указать на необходимость отобразить на экране одну обратную косую черту. Печать двойных кавычек также представляет определенную проблему для **printf**, поскольку обычно подразумевается, что двойные кавычки обозначают границы строки, и по этой причине сами по себе двойные кавычки не должны выводиться на печать. Используя esc-код \, мы сообщаем оператору **printf**, что необходимо вывести на печать двойную кавычку.

Правая фигурная скобка (**}**) означает, что выполнение функции **main** окончено.

Распространенная ошибка программирования 2.3

Вместо имени функции вывода **printf** в тексте программы набирают **print**.

Мы говорили, что **printf** вынуждает компьютер выполнить определенные *действия*. Выполнение любой программы представляет собой набор разнообразных действий, а также принятие *решений*. Конец этой главы посвящен обсуждению принятия решений. В третьей главе будут даны дальнейшие пояснения модели программирования *действие/решение*.

Необходимо отметить, что стандартные библиотечные функции, такие как **printf** и **scanf**, не являются частью языка программирования C. Поэтому компилятор не в состоянии обнаружить ошибки в **printf** или, например, **scanf**. Когда во время работы компилятору встречается оператор **printf**, он просто резервирует место в объектной программе для *вызова* библиотечной функции. Но

компилятор не знает, где находится библиотечная функция. Зато компоновщик знает. Поэтому, когда запускается компоновщик, он находит местоположение библиотечной функции и вставляет соответствующий вызов этой библиотечной функции в объектную программу. Теперь объектная программа имеет «законченный вид» и готова к выполнению. Часто скомпонованную программу так и называют: *исполняемой*. Если имя функции содержит ошибку, именно компоновщик ее обнаружит, так как не сумеет сопоставить имя в программе на языке C имени какой-либо из существующих в библиотеке функций.

Хороший стиль программирования 2.2

Для функции, производящей какую-либо печать, последний выводимый символ должен быть символом новой строки (**\n**). Это гарантирует, что функция оставит экранный курсор расположенным в начале новой строки. Соглашения такого рода облегчают возможность повторного использования программного кода – основной задачи в процессе усовершенствования программного обеспечения.

Хороший стиль программирования 2.3

Сдвигайте тело каждой функции на один абзацный отступ (три пробела) внутрь относительно скобок, ограничивающих тело функции. Это придает выразительность функциональной структуре программ и облегчает восприятие при чтении.

Хороший стиль программирования 2.4

Установив соглашение на величину отступа, вы предпочтете и в дальнейшем придерживаться такого стиля написания программ. Для создания отступов можно воспользоваться клавишей табуляции, но позиции табуляции могут изменяться, и мы рекомендуем или использовать позиции табуляции в полсантиметра, или вручную вводить три пробела на каждый уровень отступа.

Функция **printf** может напечатать **Welcome to C!** несколькими различными способами. Например, результат выполнения программы на рис. 2.3. такой же, что и программы на рис. 2.1. Дело в том, что каждая последующая функция **printf** возобновляет печать с того самого места, на котором остановилась предыдущая функция **printf**. Первая функция **printf** печатает **Welcome** и следующий за ним пробел, вторая функция **printf** начинает печатать в позиции, следующей сразу за пробелом.

Один оператор **printf** может напечатать несколько строк, если использовать символы перехода на новую строку, как показано на рис. 2.4. Каждый раз, когда встречается esc-код **\n** (новая строка), функция **printf** переводит курсор на начало следующей строки.

```
/* Печать в одну строку двумя вызовами printf */

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

```
Welcome to C!
```

Рис. 2.3. Печать в одну строку несколькими операторами **printf**

```
/* Печать нескольких строк одним вызовом printf */

main()
{
   printf("Welcome\nto\nC!\n");
}
```

```
Welcome
to
C!
```

Рис. 2.4. Печать нескольких строк одним оператором **printf**

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

Следующая программа, которую мы сейчас рассмотрим, использует стандартную библиотечную функцию **scanf**, чтобы считать два целых числа, введенные пользователем с клавиатуры, вычислить сумму их значений и напечатать результат, используя функцию **printf**. Текст программы и образец вывода представлены на рис. 2.5.

```
/* Программа сложения */
#include <stdio.h>

main()
{
   int integer1, integer2, sum;      /* объявление */

   printf("Enter first integer\n");  /* подсказка */
   scanf("%d", &integer1);           /* прочитать целое */
   printf("Enter second integer\n"); /* подсказка */
   scanf("%d", &integer2);           /* прочитать целое */
   sum = integer1 + integer2;        /* присвоить сумму */
   printf("Sum is %d\n", sum);       /* напечатать сумму */

   return 0; /* показывает успешное завершение программы */
}
```

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

Рис. 2.5. Программа суммирования

Комментарий **/* Программа сложения */** объясняет назначение программы. Строка

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

является директивой для *препроцессора* C. Строка, начинающаяся символом #, выполняется препроцессором до того, как программа начнет компилировать-

ся. Эта специфическая строка сообщает препроцессору, что необходимо включить в программу содержание *стандартного заголовочного файла ввода/вывода* (**stdio.h**). Этот заголовочный файл содержит информацию и объявления, используемые компилятором во время компиляции вызовов стандартных функций ввода/вывода, таких, как **printf**. Кроме того, заголовочный файл содержит информацию, которая помогает компилятору определить, корректно ли написаны обращения к библиотечным функциям. Более детально содержимое заголовочных файлов будет рассмотрено в главе 5.

Хороший стиль программирования 2.5

Хотя включение в текст программы **<stdio.h>** не является обязательным, это необходимо делать для каждой программы С, использующей любые стандартные функции ввода/вывода. При этом компилятор поможет вам обнаружить ошибки еще на стадии компиляции, а не на стадии исполнения (когда исправить ошибки значительно труднее).

Как было отмечено выше, каждая программа начинает исполняться с функции **main**. Левая фигурная скобка отмечает начало тела **main**, а правая соответственно конец. Строка

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

является *объявлением*. Группы символов **integer1**, **integer2** и **sum** — имена *переменных*. Переменная — это ячейка памяти, в которую можно записывать значение, предназначенное для использования программой. Приведенное выше объявление сообщает, что переменные **integer1**, **integer2** и **sum** принадлежат к типу ***int*** и, следовательно, в этих переменных будут храниться *целые* величины, то есть такие числа, как 7, –11, 0, 31914 и им подобные. Прежде чем они смогут быть использованы в программе, сразу же после левой фигурной скобки, с которой начинается тело **main**, для всех переменных должны быть объявлены имя и тип данных. Кроме **int** в языке С существуют и другие типы данных. Допускается объявление нескольких переменных одного и того же типа в одном операторе. Конечно, мы могли написать три отдельных объявления для каждой переменной, но представленный выше вариант более выразителен.

Хороший стиль программирования 2.6

Вставляйте пробел после каждой запятой (,), это облегчит восприятие текста программы.

Именем переменной в С может служить любой допустимый *идентификатор*. Идентификатор — это последовательность символов, включающая в себя буквы, цифры и символ подчеркивания (_). При этом она не должна начинаться с цифры. На длину идентификатора не существует никаких ограничений, однако согласно требованиям, установленным для языка С стандартом ANSI, лишь 31 символ распознается компилятором. Язык С различает регистр, то есть буквы верхнего и нижнего регистра воспринимаются в С по-разному, поэтому **a1** и **A1** являются различными идентификаторами.

Распространенная ошибка программирования 2.4

Использование заглавных букв там, где необходимо использовать строчные (например, вводят **Main** вместо **main**).

Совет по переносимости программ 2.1

Рекомендуется использовать идентификаторы длиной не больше 31 символа. Это гарантирует переносимость и даст возможность избежать некоторых трудно обнаружимых ошибок в процессе программирования.

Хороший стиль программирования 2.7

Осмысленный выбор имен переменных поможет сделать программу самодокументированной, то есть уменьшить количество необходимых комментариев.

Хороший стиль программирования 2.8

Первая буква идентификатора, представляющего простую переменную, должна быть набрана в нижнем регистре. Кроме того, в книге будет придаваться особый смысл идентификаторам, которые начинаются с заглавной буквы, и идентификаторам, в которых используются исключительно заглавные буквы.

Хороший стиль программирования 2.9

Имена переменных, состоящие из нескольких слов, помогут сделать программу более легкой для восприятия. Однако следует избегать слитного написания отдельных слов, как например **totalcommissions**. Лучше разделять слова при помощи символа подчеркивания **total_commissions**, или, если вы все же хотите писать слова слитно, начинайте каждое слово после первого с заглавной буквы, например, **totalCommissions**.

Объявления должны располагаться после левой фигурной скобки функции и перед *первым* исполняемым оператором. Например, в программе на рис. 2.5 объявление, помещенное после первого **printf**, должно вызвать появление сообщения о синтаксической ошибке. Сообщение о *синтаксической ошибке* появляется, когда компилятор не может распознать оператор. Компилятор в этом случае выдает сообщение, чтобы помочь программисту, найти и исправить неправильный оператор. Синтаксические ошибки — это нарушения правил языка. Часто синтаксические ошибки называют *ошибками компиляции* или *ошибками времени компиляции*.

Распространенная ошибка программирования 2.5

Объявление переменных между двумя исполняемыми операторами.

Хороший стиль программирования 2.10

Разделяйте объявления и исполняемые операторы одной пустой строкой, отмечая, таким образом, где кончаются объявления, а где начинаются исполняемые операторы.

Оператор

```
printf("Enter first integer\n");
```

печатает на экране символы **Enter first integer** и переводит курсор на начало следующей строки. Такое сообщение называется *приглашением* или *подсказкой*, так как предлагает пользователю произвести определенные действия.

Оператор

```
scanf("%d", &integer1);
```

вызывает **scanf**, чтобы *получить от пользователя некое значение*. Функция **scanf** считывает данные со стандартного устройства ввода, которым обычно является клавиатура. В нашем случае функция **scanf** имеет два аргумента, **%d** и **&integer1**. Первый аргумент — управляющая строка, задает *формат считывания*, тем самым определяется тип данных, которые предстоит ввести пользователю. Так, в частности, **%d** — *спецификация преобразования*, означающая, что вводимые данные должны быть целым числом (буква **d** используется для «десятичных целых»). Знак % в данном контексте трактуется **scanf** (в дальнейшем мы увидим, что это в полной мере относится и к **printf**), как esc-код (подобно \), а комбинация **%d** является esc-кодом (подобно **\n**). Второй аргумент **scanf** начинается со знака амперсанда (&), которым в C задается *операция взятия адреса* следующей за ним переменной. Амперсанд, когда он используется совместно с именем переменной, сообщает **scanf** ячейку памяти, в которой хранится переменная **integer1**. В последующем компьютер будет хранить величину для **integer1** в этой ячейке. Использование амперсанда очень часто смущает начинающих программистов, или людей, программировавших до этого на других языках. На данный момент просто запомните, что в операторе **scanf** необходимо имя каждой переменной предварять знаком амперсанда. Некоторые исключения из этого правила будут обсуждаться в главах 6 и 7. Настоящий смысл амперсанда станет ясным после того, как в главе 7 мы изучим указатели.

Когда компьютер выполняет вышеприведенный оператор **scanf**, он ждет, пока пользователь не введет значение для переменной **integer1**. Пользователь вводит целое число, после чего нажимает *клавишу Return* (иногда она называется *Enter*), посылая, таким образом, число в компьютер. После этого компьютер присваивает данное число, или *значение*, переменной **integer1**. Любая последующая ссылка в программе на **integer1** будет представлять именно это значение. Функции **printf** и **scanf** облегчают взаимодействие между пользователем и компьютером. Ввиду того, что подобное взаимодействие походит на диалог, часто такой режим работы называют *диалоговым* или *интерактивным*.

Оператор

```
printf("Enter second integer\n");
```

печатает на экране сообщение **Enter second integer**, после чего переводит курсор на начало следующей строки. Он также предлагает пользователю произвести определенные действия.

Оператор

```
scanf("%d", &integer2);
```

получает от пользователя значение для переменной **integer2**. В результате выполнения *оператора присваивания*

```
sum = integer1 + integer2;
```

вычисляется сумма переменных **integer1** и **integer2**, и ее значение присваивается переменной **sum** посредством *операции присваивания* =. Оператор читается как «**sum** *получает* значение **integer1 + integer2**». Большинство вычислений выполняются при помощи операции присваивания. Операция = и операция + называются *двухместными операциями* потому, что каждая имеет по два *операнда*. В случае операции + двумя операндами являются **integer1** и **integer2**. А в случае операции = двумя операндами являются **sum** и значение выражения **integer1 + integer2**.

Хороший стиль программирования 2.11

Оставлять пробелы с каждой стороны двухместной операции. Это выделит операции и придаст программе более ясный вид.

Распространенная ошибка программирования 2.6

Вычисления в операторе присваивания должны находиться с правой стороны операции =. Вычисления, помещенные с левой стороны операции присваивания, считаются синтаксической ошибкой.

Оператор

```
printf("Sum is %d\n", sum);
```

вызывает функцию **printf**, чтобы напечатать на экране текст **Sum is**, после которого следует численное значение переменной **sum**. В данном случае **printf** имеет два аргумента, "**Sum is %d\n**" и **sum**. Первый аргумент — управляющая строка, определяет формат вывода. Он содержит несколько символов, которые будут отображены, и спецификатор преобразования **%d**, определяющий, что будет напечатано целое число. Второй аргумент задает значение, которое будет напечатано. Отметим, что спецификатор преобразования для целого одинаков как для **printf**, так и для **scanf**. Это справедливо для большинства типов данных в C.

Вычисления могут выполняться и непосредственно внутри оператора **printf**. Мы могли бы скомбинировать два предыдущих оператора в один:

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

Оператор

```
return 0;
```

передает значение **0** среде операционной системы, в которой исполнялась программа. Для операционной системы это означает, что программа завершена успешно. Чтобы узнать, как сообщить о каких-либо сбоях в программе, изучите руководство по используемой вами операционной системе.

Правая фигурная скобка означает, что функция **main** окончена.

Распространенная ошибка программирования 2.7

Иногда забывают одну или обе двойные кавычки, в которые следует помещать управляющую строку в **printf** или **scanf**.

Распространенная ошибка программирования 2.8

Забывают знак **%** в спецификации преобразования в управляющей строке **printf** или **scanf**.

Распространенная ошибка программирования 2.9

Помещают esc-код, например **\n**, снаружи управляющей строки **printf** или **scanf**.

Распространенная ошибка программирования 2.10

Задают в **printf** спецификации преобразования и забывают включить в оператор выражения, значения которых должны быть напечатаны.

Распространенная ошибка программирования 2.11

Не включают в управляющую строку **printf** спецификацию преобразования, когда необходимо напечатать выражение.

Распространенная ошибка программирования 2.12

Помещают внутрь управляющей строки формата запятую, которая должна разделять управляющую строку и выражения, которые должны быть напечатаны.

Распространенная ошибка программирования 2.13

Забывают поставить перед переменной в операторе **scanf** знак амперсанда.

На многих системах эта ошибка в процессе исполнения программы называется «ошибкой сегментации» или «нарушением доступа». Подобная ошибка происходит, когда программа пытается получить доступ к той части памяти компьютера, к которой пользовательские программы доступа не имеют. Более точно причина подобной ошибки будет объяснена в главе 7.

Распространенная ошибка программирования 2.14

Перед именем переменной включенной в **printf** ставится амперсанд, тогда как на самом деле переменная не должна предваряться этим знаком.

В главе 7 мы будем изучать указатели, и рассмотрим случаи, в которых будет необходимо предварять имя переменной знаком амперсанда, чтобы напечатать адрес этой переменной. Однако на протяжении нескольких последующих глав операторы **printf** не должны содержать знаки амперсанда.

2.4. Общие понятия о памяти компьютера

Имена переменных, такие как **integer1**, **integer2** и **sum**, в действительности соответствуют *ячейкам* в памяти компьютера. Каждая переменная имеет *имя, тип* и *значение*.

В программе суммирования на рис. 2.5, во время исполнения оператора

```
scanf("%d", &integer1);
```

значение, введенное пользователем, помещается в ячейку памяти, которой присвоено имя **integer1**. Предположим, что пользователь вводит в качестве значения **integer1** число **45**. Компьютер помещает **45** в ячейку **integer1**, как показано на рис. 2.6.

Всякий раз, когда значение помещается в ячейку памяти, оно переписывает предыдущее значение, находившееся в этой ячейке. Так как предыдущая информация уничтожается, процесс считывания информации в ячейку памяти называется *разрушающим считыванием в* ячейку.

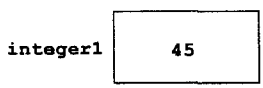


Рис. 2.6. Ячейка памяти, указывающая имя и значение переменной

Еще раз вернемся к программе сложения, когда выполняется оператор

```
scanf("%d", &integer2);
```

Предположим, пользователь вводит значение **72**. Оно помещается в ячейку **integer2**, и как теперь выглядит память, показано на рис. 2.7. Заметим, что ячейки памяти совсем не обязательно окажутся соседними.

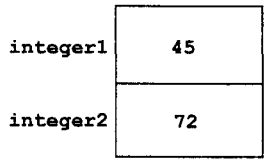


Рис. 2.7. Ячейки памяти после ввода значений для обеих переменных

После того, как программа получила значения для **integer1** и **integer2**, она складывает эти величины и помещает суммарное значение в переменную **sum**. Оператор, выполняющий сложение

```
sum = integer1 + integer2;
```

тоже влечет за собой разрушающее считывание в ячейку. Это происходит, когда вычисленная сумма **integer1** и **integer2** помещается в ячейку **sum** (стирая при этом значение, которое могло уже находиться в **sum**). После того как **sum** вычислена, память выглядит, как показано на рис. 2.8. Отметим, что значения **integer1** и **integer2** после использования в вычислении **sum** никак не изменились. Эти величины использовались, но не были уничтожены во время выполнения вычислений компьютером. Таким образом, когда значение считывается из ячейки памяти, то такой процесс называется *неразрушающим считыванием из* ячейки.

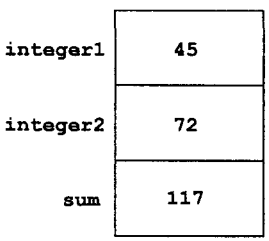


Рис. 2.8. Ячейки памяти после вычисления

2.5. Арифметика в C

Большинство написанных на языке C программ выполняют арифметические вычисления. Сводная таблица *арифметических операций* языка C представлена на рис. 2.9. Отметим использование различных специальных символов, не используемых в алгебре. Так, *звездочка* (*) означает умножение, а *знак процента* (%) означает операцию *взятия по модулю*, которая будет объяснена ниже. В алгебре, если мы хотим умножить *a* на *b*, мы можем просто написать переменные, обозначенные одной буквой, рядом, то есть *ab*. Однако если мы напишем нечто подобное в C, то конструкция **ab** будет воспринята языком, как одна переменная

(или идентификатор) из двух букв. Таким образом, C (как, впрочем, и многие другие языки программирования) требует, чтобы умножение было явно обозначено с использованием знака звездочки следующим образом: **a * b**.

Действие в C	Арифметическая операция	Алгебраическое выражение	Выражение на C
Сложение	+	f + 7	f + 7
Вычитание	–	p – 7	p – c
Умножение	*	bm	b * m
Деление	/	x/y или $\frac{x}{y}$ или x÷y	x / y
Взятие по модулю	%	r mod s	r % s

Рис. 2.9. Арифметические операции

Все арифметические операции являются двухместными. Например, выражение **3 + 7** содержит двухместную операцию + и операнды **3** и **7**.

Результатом деления двух *целых* чисел также будет целое число. Например, значение выражения **7 / 4** будет **1**, а **17 / 5** равно **3**. В C существует операция взятия по модулю, %, которая дает значение остатка после деления нацело. Операция взятия по модулю может использоваться только с целыми операндами. Выражение **x % y** означает остаток от деления **x** на **y**. Таким образом, результатом **7 % 4** является **3**, а **17 % 5** соответственно **2**. В дальнейшем будет рассмотрено много интересных вариантов применения операции взятия по модулю.

Распространенная ошибка программирования 2.15

Деление на ноль обычно не определено в компьютерных системах и, как правило, приводит к фатальной ошибке, то есть такой ошибке, в результате которой выполнение программы немедленно прерывается. В случае не фатальной ошибки программа выполнится до конца, однако результат выполнения программы, как правило, неверен.

Арифметические выражения в C должны записываться в строчку, в таком виде легче ввести программу в компьютер. Таким образом, выражения типа «**a** деленное на **b**» следует записывать как **a/b**, чтобы все операции и операнды располагались на одной строке. Алгебраическая нотация

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

неприемлема, как правило, для компиляторов, хотя существуют созданные для специальных целей пакеты программного обеспечения, поддерживающие более привычную нотацию сложных математических выражений.

Круглые скобки используются в выражениях на языке C точно так же, как и в алгебраических выражениях. Например, чтобы умножить **a** на **b+c** мы пишем:

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

C *оценивает* арифметические выражения (т.е. вычисляет их численное значение) в последовательности, определяемой *правилами старшинства операций*, которые в общем такие же, как и в алгебре:

1. Выражения или части выражений, находящиеся внутри скобок, оцениваются в первую очередь. Другими словами, *могут использоваться скобки, чтобы поменять порядок вычислений на тот, что требуется программисту.* Говорят, что *скобки* обладают «наивысшим приоритетом». В случае вложенных скобок выражение, находящееся во внутренней паре скобок, оценивается первым.

2. Следующими выполняются операции умножения, деления и взятия по модулю. Если в каком-нибудь выражении содержится несколько операций умножения, деления или взятия по модулю, оценка производится слева направо. Говорят, что умножение, деление и взятие по модулю — операции *одинакового приоритета.*

3. Последними выполняются операции сложения и вычитания. Если выражение содержит несколько операций сложения и вычитания, оценка производится слева направо. Сложение и вычитание также операции одного приоритета.

Правила, определяющие приоритет операций, являются тем руководящим принципом, который позволяет С правильно оценивать выражения. Когда оценка происходит слева направо, говорят об *ассоциативности слева направо.* Мы увидим, что некоторые операции ассоциативны справа налево. На рис. 2.10 представлена сводная таблица старшинства операций в порядке убывания приоритета.

Теперь рассмотрим несколько выражений в плане применения правил старшинства операций. *Для каждого примера дается алгебраическая запись и* равнозначная запись на С.

Операция	Действие	Порядок оценки (приоритет)
()	Скобки	Выражения в скобках оцениваются в первую очередь. В случае вложенных скобок в первую очередь вычисляется значение во внутренних скобках. А в случае нескольких пар скобок «одного уровня» (то есть не вложенных) они вычисляются слева направо.
*, / или %	Умножение, деление, взятие по модулю	Оцениваются вслед за скобками. Если операций несколько, то вычисления ведутся слева направо.
+ или -	Сложение или вычитание	Оцениваются в последнюю очередь. Если операций несколько, то вычисления ведутся слева направо.

Рис. 2.10. Старшинство арифметических операций

Следующий пример вычисляет среднее арифметическое для пяти элементов:

Алгебра: $m = \dfrac{a+b+c+d+e}{5}$

```
С:      m = (a + b + c + d + e) / 5;
```

Скобки необходимы по той причине, что операция деления обладает более высоким приоритетом, чем сложение. Необходимо поделить сумму **(a + b + c + d + e)** на **5**. Если скобки по ошибке пропущены, мы получим **a + b + c + d + e / 5**, что приведет к вычислению совсем другого выражения

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

Еще один пример — уравнение прямой линии:

Алгебра: $y = mx + b$

```
С:        y = m * x + b;
```

В данном случае скобки не требуются. Умножение выполняется первым, так как операция умножения имеет более высокий приоритет, чем операция сложения.

Следующий пример содержит операции взятия по модулю (%), умножения, деления, сложения и вычитания:

Алгебра: $z = pr \bmod q + w / x - y$

```
С:        z = p * r % q + w / x - y;
                  (1)   (2)   (4)   (3)   (5)
```

Обведенные кружками числа, расположенные под оператором, показывают порядок, в котором С производит вычисления. Умножение, взятие по модулю и деление оцениваются в первую очередь в порядке слева направо (они ассоциативны слева направо), так как имеют более высокий приоритет, чем сложение и вычитание. Следующими выполняются операции сложения и вычитания, они также оцениваются слева направо.

Не все выражения, в которых есть несколько пар скобок, содержат вложенные скобки. Выражение

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

не содержит вложенных скобок. Говорят, что в данном случае скобки находятся на «одном уровне». В этом случае С вычисляет заключенные в скобки выражения в первую очередь, порядок вычисления — слева направо.

Чтобы лучше понять правила старшинства операций, рассмотрим, как С вычисляет полином второй степени.

```
z = a * x * x + b * x + c;
        (1)   (2)   (4)   (3)   (5)
```

Обведенные кружками числа, расположенные под оператором, показывают порядок, в котором С производит оценку. В языке С нет арифметической операции возведения в степень. Поэтому мы представили x^2 как **x * x**. Стандартная библиотека С включает в себя функцию **pow** возведения в степень. По причине того, что существуют некоторые тонкости применения **pow**, связанные с типом требуемых функцией данных, мы отложим детальное объяснение ее до главы 4.

Предположим, что **a = 2, b = 3, c = 7** и **x = 5**. Рис. 2.11 иллюстрирует, как будет вычисляться значение представленного выше полинома второй степени.

2.6. Принятие решений: операции равенства и отношения

Исполняемые операторы С или выполняют действия (такие, например, как вычисления или ввод-вывод данных), или принимают *решения* (скоро мы увидим несколько примеров этого). Мы можем принять решение в программе,

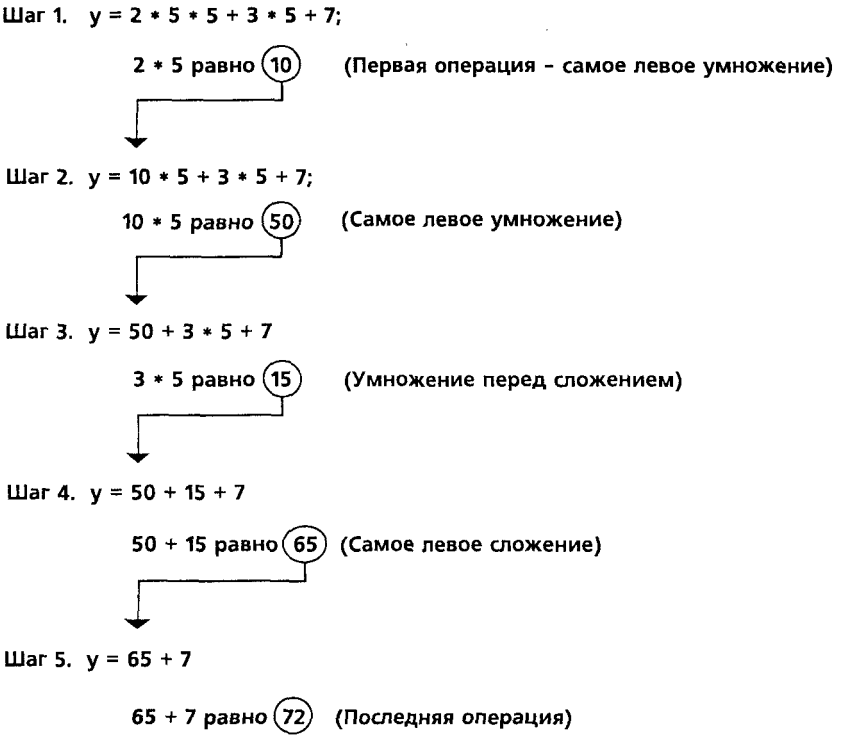


Рис. 2.11. Вычисление полинома второй степени

например: определить отметку некоторого человека на экзамене, и если она окажется выше или равна 60, напечатать сообщение «Поздравляем! Вы прошли». В этом разделе приведен упрощенный вариант *управляющей структуры* **if** языка C, который позволяет программе принимать решения исходя из того, истинно или ложно некое утверждение, называемое *условием*. Если условие выполняется (т.е. оно *истинно*), тело структуры **if** исполняется. Если же условие не выполняется (т.е. оно *ложно*), то тело оператора не исполняется. Вне зависимости от того, исполнилось тело оператора или нет, после завершения выполнения оператора **if** выполняется следующий за ним оператор.

Условия в операторе **if** задаются с использованием операций *равенства* и *отношения*, сводная таблица которых представлена на рис. 2.12. Операции отношения имеют одинаковый приоритет и ассоциативны слева направо. Операции равенства имеют более низкий приоритет и также ассоциативны слева направо. (Замечание: в C условием может какое-либо выражение, значение которого либо равно нулю (false), либо нет (true). На всем протяжении книги нам будут встречаться многочисленные примеры подобного способа задавать условие в операторе **if**.)

Распространенная ошибка программирования 2.16

Разделение символов в любой из операций ==, !=, >= или <= пробелами является синтаксической ошибкой.

Стандартная алгебраическая операция равенства или отношения	Операция равенства или отношения в С	Пример условия в С	Смысл условия в С
Операции равенства			
=	==	x == y	x равен y
≠	!=	x != y	x не равен y
Операции отношения			
>	>	x > y	x больше y
<	<	x < y	x меньше y
≥	>=	x >= y	x больше или равен y
≤	<=	x <= y	x меньше или равен y

Рис. 2.12. Операции равенства и отношения

Распространенная ошибка программирования 2.17

Два символа в любой из операций **!=**, **>=** или **<=**, набранные в обратном порядке (соответственно **=!**, **=>** и **=<**), также приведут к синтаксической ошибке.

Распространенная ошибка программирования 2.18

Часто путают операцию **==** с операцией присваивания **=**.

Чтобы избежать этой ошибки, следует читать знак операции равенства как «двойное равенство», а знак операции присваивания как «присвоить значение». Как мы увидим в дальнейшем, путаница в применении этих операций не всегда приводит к появлению легко распознаваемой синтаксической ошибки, а, напротив, может быть причиной чрезвычайно трудно обнаружимых логических ошибок.

Распространенная ошибка программирования 2.19

Иногда ставят точку с запятой непосредственно за правой скобкой, завершающей выражение условия структуры **if**.

Пример на рис. 2.13 использует шесть операторов **if**, чтобы сравнить два введенных пользователем числа. Если условие в любом из операторов **if** истинно, то выполняется оператор **printf**, связанный с данным **if**. Программа и три варианта вывода в результате ее выполнения показаны на рисунке.

Заметим, что программа на рис. 2.13 использует **scanf**, чтобы ввести два числа. Каждой спецификации преобразования соответствует аргумент, в котором будет сохраняться введенное значение. Первая спецификация **%d** преобразует значение, которое должно сохраняться в переменной **num1**, а вторая спецификация **%d** преобразует значение, которое будет сохраняться в переменной **num2**. Смещая вправо тело каждого оператора **if** и помещая пустые строки сверху и снизу от него, мы облегчаем восприятие программы. Заметим также, что каждый оператор **if** на рис. 2.13. содержит в своем теле единствен-

ный оператор. В главе 3 будет показано, как записать оператор **if**, тело которого содержит несколько операторов.

```
/* Применение операторов if,
   операций отношения и операций равенства */
#include <stdio.h>

main()
{
   int num1, num2;

   printf("Enter two integers, and I will tell you\n");
   printf("the relationships they satisfy: ");
   scanf("%d%d", &num1, &num2);  /* прочитать два целых */

   if (num1 == num2)
      printf("%d is equal to %d\n", num1, num2);

   if (num1 != num2)
      printf("%d is not equal to %d\n", num1, num2);

   if (num1 < num2)
      printf("%d is less than %d\n", num1, num2);

   if (num1 > num2)
      printf("%d is greater than %d\n", num1, num2);

   if (num1 <= num2)
      printf("%d is less than or equal to %d\n", num1, num2);

   if (num1 >= num2)
      printf("%d is greater than or equal to %d\n", num1, num2);

   return 0;  /* показывает успешное завершение программы */
}
```

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

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

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

Рис. 2.13. Использование операций равенства и отношения

Хороший стиль программирования 2.12

Смещать вправо операторы в теле **if**.

Хороший стиль программирования 2.13

Помещать в программе пустую строку над и под каждым оператором **if** для облегчения восприятия программы.

Хороший стиль программирования 2.14

В каждой строке программы должно быть не более одного оператора.

Распространенная ошибка программирования 2.20

Ставят запятые (хотя они не нужны) между спецификациями преобразования в управляющей строке формата считывания оператора **scanf**.

Комментарий к программе на рис. 2.13 разбит на две строки. В программах на языке C *пробельные символы*, такие как табуляции, новые строки, пробелы обычно игнорируются. Поэтому можно разбивать операторы и комментарии на несколько строк. Однако идентификаторы разбивать нельзя.

Таблица на рис. 2.14 показывает старшинство операций, введенных в этой главе. Операции расположены сверху вниз в порядке убывания приоритета. Заметим, что знак равенства тоже является операцией. Все эти операции за *исключением операции присваивания* = *ассоциативны слева направо. Операция* присваивания ассоциативна справа налево.

Хороший стиль программирования 2.15

Длинные операторы можно записывать в несколько строк. Если оператор необходимо разбить на несколько строк, старайтесь делать это осмысленно (скажем, после запятой в разделенном запятыми списке). Если оператор разбит на две или большее число строк, смещайте вправо все следующие за первой строки.

Хороший стиль программирования 2.16

При написании выражений, содержащих много операций, пользуйтесь таблицей старшинства операций. Следите за тем, чтобы операции в выражении выполнялись в правильном порядке. Если вы не уверены в порядке выполнения операций в сложном выражении, используйте скобки для принудительного задания нужного порядка, так же как делаете это в алгебраических выражениях. Не следует также забывать, что некоторые операции в C, например операция присваивания (=), ассоциативны справа налево, а не наоборот.

Некоторые из слов, которые мы использовали при написании программ на C в этом параграфе — в частности **int**, **return** и **if** — являются *ключевыми* или *зарезервированными* словами языка. Полный набор ключевых слов языка C приведен на рис. 2.15. Эти слова имеют специальное значение для компилятора C, поэтому программист должен быть внимателен и не применять эти слова как идентификаторы имен переменных. В книге будут рассмотрены все ключевые слова C.

Операция	Ассоциативность
()	слева направо
* / %	слева направо
+ -	слева направо
< <= > =>	слева направо
== !=	слева направо
=	справа налево

Рис. 2.14. Старшинство и ассоциативность рассмотренных на данный момент операций

В данной главе мы познакомились с целым рядом возможностей языка программирования C, включая вывод данных на экран, ввод данных пользователем, выполнение вычислений и принятие решений. В следующей главе, основываясь на пройденном в этой главе материале, мы введем понятие структурного программирования. Вы подробнее познакомитесь с принципами применения отступов; узнаете, как задавать порядок, в котором должны выполняться операторы — т.е. контролировать то, что называется *потоком управления*.

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

Рис. 2.15. Зарезервированные слова

Резюме

- Комментарии начинаются с /* и заканчиваются */. Программисты вставляют комментарии для документирования программ и облегчения их восприятия. Комментарии не оказывают никакого влияния на работу компьютера во время исполнения программы.
- Директива препроцессора **#include <stdio.h>** сообщает компилятору, что необходимо включить в программу стандартный заголовочный файл ввода/вывода. Этот файл содержит информацию, которая помогает компилятору определить, корректно ли написаны обращения к функциям ввода-вывода, например, **scanf** и **printf**.
- Программа C состоит из функций, одна из которых обязательно должна быть **main**. Каждая программа начинает выполняться с функции **main**.

- Функция **printf** может использоваться для вывода на печать заключенной в кавычки строки и значений выражений. При печати целого значения первый аргумент функции **printf** — управляющая строка, задающая формат вывода, должна содержать спецификатор преобразования **%d** и, кроме того, может содержать любые другие символы, которые должны быть напечатаны; второй аргумент — выражение, значение которого необходимо напечатать. Если на печать будет выводиться более одного целого числа, управляющая строка формата должна содержать **%d** для каждого целого, а разделенные запятой аргументы, следующие за управляющей строкой, содержать выражения, значения которых должны быть напечатаны.
- Функция **scanf** получает значения, которые пользователь обычно вводит с клавиатуры. Ее первый аргумент — управляющая строка, задающая формат считывания, которая сообщает компьютеру, какой тип данных должен быть введен пользователем. Спецификация преобразования **%d** определяет, что эти данные должны быть целыми. Каждому из оставшихся аргументов должна соответствовать спецификация преобразования в управляющей строке формата считывания. Имя каждой переменной обычно предваряется амперсандом (&), называемым в С операцией взятия адреса. Амперсанд при объединении с именем переменной сообщает компьютеру ячейку памяти, в которой будет храниться значение. Затем компьютер записывает значение в эту ячейку.
- Все переменные в программе на С должны быть объявлены, прежде чем они будут использованы в программе.
- Именем переменной в С может быть любой допустимый идентификатор. Идентификатор представляет собой последовательность символов, состоящую из букв, цифр и символа подчеркивания (_). Идентификаторы не могут начинаться с цифры. Идентификаторы могут быть любой длины; однако согласно стандарту ANSI значим лишь тридцать один символ.
- Язык С различает регистр.
- Большинство вычислений выполняются при помощи операторов присваивания.
- Каждая переменная, хранящаяся в памяти компьютера, имеет имя, значение и тип.
- Всякий раз, когда новое значение помещается в ячейку памяти, оно заменяет предыдущее значение, находившееся в этой ячейке. Так как эта предыдущая информация уничтожается, процесс помещения информации в ячейку памяти назван разрушающим считыванием в ячейку.
- Процесс считывания значения из памяти называют неразрушающим считыванием из ячейки.
- Чтобы облегчить введение программы в компьютер, арифметические выражения в С должны записываться в строчку.
- С вычисляет арифметические выражения в точном соответствии с правилами, определяющими приоритет и ассоциативность операций.
- Оператор **if** позволяет программисту вводить в программу возможность выбора, в зависимости от выполнения некоторого условия. Формат оператора **if** следующий:

if *(условие)*
оператор

Если условие истинно, оператор в теле **if** выполняется. Если условие ложно, тело оператора пропускается.

- Условия в операторе **if** обычно задаются с использованием операций равенства и отношения. Результат этих операций всегда «истина» или «ложь». Заметим, что условием может быть также любое выражение, порождающее нулевое (false) или ненулевое (true) значение.

Терминология

C
esc-код
esc-символ
false
int
main
stdio.h
true
амперсанд (**&**)
аргумент
арифметические операции
ассоциативность операций
ассоциативность слева направо
ассоциативность справа налево
блок
вложенные скобки
двухместные операции
действие
деление на 0
деление нацело
диалоговое вычисление
запись в строчку
зарезервированные слова
зарезервированные слова языка C
звездочка (*****)
знак процента % (esc-символ)
знак равенства =
(операция присваивания)
значение
значение переменной
идентификатор
имя
интерактивное вычисление
истинность
клавиша enter
клавиша return
ключевые слова
комментарий
круглые скобки **()**
литерал
ложность
модель действие/решение
не равное нулю **(true)**
не фатальная ошибка
неразрушающее чтение из ячейки
нуль **(false)**
обратная косая черта \ (esc-символ)
объявление
окончание оператора (;)
операнд
оператор
оператор присваивания
операции отношения
< «меньше»
<= «меньше или равно»
> «больше»
>= «больше или равно»
операции равенства
!= «не равно»
== «равно»
операция
операция взятия адреса
операция взятия по модулю (%)
операция присваивания (=)
операция умножения (*)
отступ
ошибка времени компиляции
ошибка при трансляции
память
переменная
правила старшинства операций
препроцессор C
приглашение
принятие решения
приоритет операций
пробельные символы
различение регистра
разрушающее считывание в ячейку
решение
символ новой строки (**\n**)
символ подчеркивания (_)
синтаксическая ошибка
сообщение
спецификация преобразования
спецификация преобразования **%d**
стандартная библиотека C
стандартный заголовочный файл
ввода/вывода

строка
строка символов
структурное программирование
тело функции
тип переменной
точка с запятой ;
(окончание оператора)
управление, поток управления
управляющая строка
управляющая строка формата
управляющая структура `if`
условие
фатальная ошибка
фигурные скобки {}
функция
функция `printf`
функция `scanf`
целое
ячейка
ячейка памяти

Распространенные ошибки программирования

2.1. Часто забывают закончить комментарий символами */.

2.2. Часто начинают комментарий символами */, а заканчивают символами /*.

2.3. Вместо имени функции вывода **printf** в тексте программы набирают **print**.

2.4. Использование заглавных букв там, где должны быть использованы строчные (например, вводят **Main** вместо **main**).

2.5. Объявление переменных между двумя исполняемыми операторами.

2.6. Вычисления в операторе присваивания должны находиться с правой стороны операции =. Вычисления, помещенные с левой стороны операции присваивания, считаются синтаксической ошибкой.

2.7. Иногда забывают одну или обе двойные кавычки, в которые следует помещать управляющую строку в **printf** или **scanf**.

2.8. Забывают знак % в спецификации преобразования в управляющей строке **printf** или **scanf**.

2.9. Помещают esc-код, например **\n**, снаружи управляющей строки формата **printf** или **scanf**.

2.10. Задают в **printf** спецификации преобразования и забывают включить в оператор выражения, значения которых должны быть напечатаны.

2.11. Не включают в управляющую строку **printf** спецификацию преобразования, когда необходимо напечатать выражение.

2.12. Помещают внутрь управляющей строки формата запятую, которая должна разделять управляющую строку и выражения, которые должны быть напечатаны.

2.13. Забывают поставить перед переменной в операторе **scanf** знак амперсанда.

2.14. Перед именем переменной включенной в **printf** ставится амперсанд, тогда как на самом деле переменная не должна предваряться этим знаком.

2.15. Деление на ноль обычно не определено в компьютерных системах и, как правило, приводит к фатальной ошибке, то есть такой ошибке, в

результате которой выполнение программы немедленно прерывается. В случае не фатальной ошибки программа выполнится до конца, однако результат выполнения программы, как правило, неверен.

2.16. *Разделение символов в любой из операций* ==, !=, >= *или* <= *пробелами является синтаксической ошибкой.*

2.17. Два символа в любой из операций !=, >= или <=, набранные в обратном порядке (соответственно =!, => и =<), также являются синтаксической ошибкой.

2.18. *Часто путают операцию* == *с операцией присваивания* =.

2.19. Иногда ставят точку с запятой непосредственно за правой скобкой, завершающей выражение условия структуры **if**.

2.20. Ставят запятые (хотя они не нужны) между спецификациями преобразования в управляющей строке формата считывания оператора **scanf**.

Хороший стиль программирования

2.1. *Каждая функция должна быть предварена комментарием, объясняющим ее назначение.*

2.2. Для функции, производящей какую-либо печать, последний выводимый символ должен быть символом новой строки (**\n**). Это гарантирует, что функция оставит экранный курсор расположенным в начале новой строки. Соглашения такого рода облегчают возможность повторного использования программного кода — основной задачи в процессе усовершенствования программного обеспечения.

2.3. Сдвигайте тело каждой функции на один абзацный отступ (три пробела) *внутрь относительно скобок, ограничивающих тело функции.* Это придает выразительность функциональной структуре программ и облегчает *восприятие при чтении.*

2.4. Установив соглашение на величину отступа, вы предпочтете и в дальнейшем придерживаться такого стиля написания программ. Для создания отступов можно воспользоваться клавишей табуляции, но позиции табуляции могут изменяться, и мы рекомендуем или использовать позиции табуляции в полсантиметра, или вручную вводить три пробела на каждый уровень отступа.

2.5. *Хотя включение в текст программы* **<stdio.h>** *не является обязательным, это необходимо делать для каждой программы С, использующей любую из стандартных функций ввода/вывода. При этом компилятор поможет вам обнаружить ошибки еще на стадии компиляции, а не на стадии исполнения (когда исправить ошибки значительно труднее).*

2.6. Вставляйте пробел после каждой запятой (,), это облегчит восприятие текста программы.

2.7. Осмысленный выбор имен переменных поможет сделать программу самодокументированной, то есть уменьшить количество необходимых комментариев.

2.8. Первая буква идентификатора, представляющего простую переменную, должна быть набрана в нижнем регистре. Кроме того, в книге будет придаваться особый смысл идентификаторам, которые начинаются с заглавной буквы, и идентификаторам, в которых используются исключительно заглавные буквы.

2.9. Имена переменных, состоящие из нескольких слов, помогут сделать программу более легкой для восприятия. Однако следует избегать слитного написания отдельных слов, как например **totalcommissions**. Лучше разделять слова при помощи символа подчеркивания **total_commissions**, или, если вы все же хотите писать слова слитно, начинайте каждое слово после первого с заглавной буквы, например, **totalCommissions**.

2.10. Разделяйте объявления и исполняемые операторы одной пустой строкой, отмечая, таким образом, где кончаются объявления, а где начинаются исполняемые операторы.

2.11. Оставлять пробелы с каждой стороны двухместной операции. Это выделит операции и придаст программе более ясный вид.

2.12. Смещать вправо операторы в теле **if**.

2.13. Помещать в программе пустую строку над и под каждым оператором **if** для облегчения восприятия программы.

2.14. В каждой строке программы должно быть не более одного оператора.

2.15. Длинные операторы можно записывать в несколько строк. Если оператор необходимо разбить на несколько строк, старайтесь делать это осмысленно (скажем, после запятой в разделенном запятыми списке). Если оператор разбит на две или большее число строк, смещайте вправо все следующие за первой строки.

2.16. При написании выражений, содержащих много операций, пользуйтесь таблицей старшинства операций. Следите за тем, чтобы операции в выражении выполнялись в правильном порядке. Если вы не уверены в порядке выполнения операций в сложном выражении, используйте скобки для принудительного задания нужного порядка, так же как делаете это в алгебраических выражениях. Не следует также забывать, что некоторые операции в С, например операция присваивания (=), ассоциативны справа налево, а не наоборот.

Советы по переносимости программ

2.1. Рекомендуется использовать идентификаторы с количеством символов, равным или меньшим 31. Это гарантирует переносимость и даст возможность избежать некоторых трудно обнаружимых ошибок в процессе программирования.

Упражнения для самоконтроля

2.1. Заполните пустые места в каждом из следующих утверждений.

a) Каждая С программа начинает выполняться с функции _____.

b) Тело каждой функции начинается с _____ и заканчивается _____.

c) Каждый оператор заканчивается _____.

d) Для вывода информации на экран используется стандартная библиотечная функция _____.

e) Esc-код **\n** представляет символ _____, вызывающий перевод курсора в начальную позицию следующей строки экрана.

f) Для приема данных с клавиатуры используется стандартная библиотечная функция _____.

g) Спецификация преобразования _____ помещается в управляющей строке формата **scanf** для того, чтобы показать, что будет вводиться целое число, и в управляющей строке формата **printf**, чтобы показать, что будет выводиться целое число.

h) Всякий раз, когда новое значение помещается в ячейку памяти, это значение переписывает предыдущее значение, находившееся в ячейке. Этот процесс известен как _____ считывание в ячейку.

i) Когда значение считывается из ячейки памяти и величина в этой ячейке сохраняется, то это называется _____ считыванием из ячейки.

j) Оператор _____ используется для того, чтобы принять решение.

2.2. Установите, являются ли следующие утверждения верными или неверными; если утверждение неверно, объясните, почему.

a) Когда вызывается функция **printf**, она всегда начинает печатать с начала новой строки.

b) Комментарии заставляют компьютер при выполнении программы выводить на экран текст, заключенный между /* и */.

c) Esc-код **\n** при использовании в управляющей строке формата функции **printf** перемещает курсор в начальную позицию следующей строки экрана.

d) Все переменные должны быть объявлены, прежде чем будут использоваться.

e) При объявлении переменной необходимо указать ее тип.

f) Язык С рассматривает переменные **number** и **NuMbEr** как тождественные.

g) Объявление можно поместить в любом месте тела функции.

h) Все аргументы функции **printf**, следующие за управляющей строкой формата, должны предваряться амперсандом (&).

i) Операция взятия по модулю (%) может использоваться только с целыми операндами.

j) Арифметические операции *, /, %, +, и - имеют одинаковый приоритет.

k) Верно ли следующее утверждение: следующие имена переменных идентичны для всех систем С, удовлетворяющих стандарту ANSI

```
thisisasuperduperlongname1234567
thisisasuperduperlongname1234568
```

l) Верно ли следующее утверждение: в программе на языке С, чтобы вывести на печать три строки, необходимо использовать три оператора **printf**.

2.3. Как с помощью одного оператора С выполнить каждое из следующих действий:

a) Объявить переменные типа **int**: **c**, **thisVariable**, **q766354** и **number**.

b) Предложить пользователю ввести целое число. Закончите ваше приглашающее сообщение двоеточием (:), за которым должен следовать пробел, и оставьте курсор в позиции непосредственно за пробелом.

c) Считать целое число, введенное с клавиатуры, и сохранить его значение в переменной **a** типа **int**.

d) Если переменная **number** не равна **7**, выдать сообщение «**The variable number is not equal to 7.**».

e) Напечатать сообщение «**This is a C program.**» в одну строку.

f) Напечатать сообщение «**This is a C program.**» в две строки так, чтобы первая строка заканчивалась на **C**.

g) Напечатать сообщение «**This is a C program.**» так, чтобы каждое слово располагалось на отдельной строке.

h) Напечатать сообщение «**This is a C program.**» так, чтобы каждое слово располагалось в новой позиции табуляции.

2.4. Напишите оператор (или комментарий), содержащий:

a) Утверждение, что программа будет вычислять произведение трех целых чисел.

b) Объявление, что переменные **x**, **y**, **z** и **result** должны принадлежать к типу **int**.

c) Приглашение пользователю ввести три целых числа.

d) Считать три целых числа с клавиатуры и сохранить их в переменных **x**, **y** и **z**.

e) Вычислить произведение трех целых чисел, значения которых содержаться в переменных **x**, **y**, **z** и присвоить результирующее значение переменной **result**.

f) Вывести на печать сообщение «**The product is**» и следом за ним значение переменной **result**.

2.5. Используя операторы, написанные в упражнении 2.4, напи те полную программу, вычисляющую произведение трех целых чисел.

2.6. Найдите и исправьте ошибки в каждом из следующих операторов:

a) `printf("The value is %d\n", &number);`

b) `scanf("%d%d",&number1,number2);`

c)
```
if (c < 7);
   printf ("C is  less than 7\n";
```

d)
```
if (c => 7)
   printf("C is equal to or less than 7\n");
```

Ответы на упражнения для самоконтроля

2.1. a) **main**. b) левая фигурная скобка ({), правая фигурная скобка (}), c) точка с запятой, d) **printf**, e) новая строка, f) **scanf**, g) **%d**, h) разрушающее, i) неразрушающее, j) **if**.

2.2. a) Неверно. Функция **printf** всегда начинает печать с того места, где находится курсор, а курсор может находиться в любой позиции строки.

b) Неверно. Комментарии не вызывают выполнения каких-либо действий во время выполнения программы. Они используются для документирования программ и для того, чтобы повысить их удобочитаемость.

c) Верно.

d) Верно.

e) Верно.

f) Неверно. C различает регистр, поэтому данные переменные воспримутся компилятором как две различные

g) Неверно. Объявления должны размещаться после левой фигурной скобки тела функции и до любого из исполняемых операторов.

h) Неверно. Аргументы функции **printf** обычно не предваряются амперсандом. Аргументы, следующие за управляющей строкой формата функции **scanf**, напротив, обычно предваряются амперсандом. Исключения из данного правила будут обсуждены в главах 6 и 7.

i) Верно.

j) Неверно. Операции *, / и % имеют одинаковый приоритет, а операции + и − более низкий приоритет.

k) Неверно. Некоторые системы могут различать идентификаторы длиной более 31 символа.

l) Неверно. Один оператор **printf** с несколькими esc-кодами **\n** может напечатать несколько строк.

2.3. a) `int c, thisVariable, q76354, number;`

b) `printf("Enter an integer: ");`

c) `scanf("%d", &a);`

d)
```
if (number != 7)
     printf("The variable number is not equal to 7.\n");
```

e) `printf("This is a C program.\n");`

f) `printf("This is a C\nprogram.\n");`

g) `printf("This\nis\na\nC\nprogram.\n");`

h) `printf("This\tis\ta\tC\tprogram.\n");`

2.4. a) `/* Calculate the product of three integers */`

b) `int x, y, z, result;`

c) `printf("Enter three integers: ");`

d) `scanf ("%d%d%d", &x, &y, &z);`

e) `result = x * y * z;`

f) `printf("The product is %d\n", result);`

2.5.
```
/* Calculate the product of three integers */
#include <stdio.h>

main()
{
    int x, y, z, result;

    printf("Enter three integers: ");
    scanf ("%d%d%d", &x, &y, &z);
    result = x * y * z;
    printf("The product is %d\n", result);

    return 0;
}
```

2.6. a) Ошибка: **&number**. Исправление: убрать значок &. Позже в книге будут рассмотрены исключения их этого правила.

b) Ошибка: перед **number2** нет амперсанда. Исправление: вместо **number2** необходимо написать **&number2**. Позже в книге будут рассмотрены исключения их этого правила.

c) Ошибка: точка с запятой после правой скобки в условии оператора **if**. Исправление: Убрать точку с запятой после правой скобки. Замечание: результатом такой ошибки является то, что оператор **printf** будет выполнен независимо от того, истинно или ложно условие в **if**. Точка с запятой после правой скобки трактуется компилятором как пустой оператор — оператор, который ничего не делает.

d) Ошибка: операцию отношения => следует заменить на >=.

Упражнения

2.7. Найдите и исправьте ошибки в каждом из следующих операторов (Замечание: в некоторых операторах ошибок может быть несколько ошибок):

a) `scanf("d", value);`

b) `printf("The product of %d and %d is %d"\n, x,y);`

c) `firstNumber + secondNumber = sumOfNumbers`

d)
```
if (number => largest)
    largest == number;
```

e) `*/ Program to determine the largest of three integers /*`

f) `Scanf("%d", anInteger);`

g) `printf("Remainder of %d divided by %d is\n", x, y, x % y);`

h)
```
if (x = y);
    printf(%d is equal to %d\n", x, y);
```

i)
```
print("The sum is %d\n," x + y);
```

j)
```
Printf("The value you enterd is: %d\n, &value);
```

2.8. Заполните пробелы в каждом из следующих утверждений:

a) _____ используется для документирования программ и повышения их удобочитаемости.

b) Для вывода информации на экран используется функция _____.

c) Оператор языка C, предназначенный для принятия решений — это _____.

d) Вычисления обычно выполняются при помощи операторов _____.

e) Функция, предназначенная для ввода данных с клавиатуры — это _____.

2.9. Напишите оператор C или строку, выполняющие следующие действия:

a) Печатает сообщение **«Enter two numbers»**.

b) Присваивает значение произведения переменных **b** и **c** переменной **a**.

c) Констатируйте, что программа представляет собой пример вычисления заработной платы (т.е. напишите текст, который помог бы документировать программу).

d) Введите три целых числа с клавиатуры и поместите их значения в целые переменные **a**, **b** и **c**.

2.10. Установите, какие из следующих утверждений верны, а какие нет. Объясните свои ответы.

a) Операции в C оцениваются слева направо.

b) Ниже перечисленные имена переменных являются допустимыми: **_under_bar_**, **m928134**, **t5**, **j7**, **her_sales**, **his_account_total**, **a**, **b**, **c**, **z**, **z2**.

c) Оператор **printf(«a = 5;»);** — типичный пример оператора присваивания.

d) Допустимое арифметическое выражение на C, не содержащее скобок, будет вычисляться слева направо.

e) Все ниже перечисленные имена переменных являются недопустимыми: **3g**, **87**, **67h2**, **h22**, **2h**.

2.11. Впишите вместо пробелов правильный ответ.

a) Какие арифметические операции имеют приоритет, одинаковый с умножением? _____.

b) В случае вложенных скобок выражение какой пары скобок будет вычисляться в первую очередь в арифметическом выражении? _____.

c) Ячейка памяти компьютера, которая в процессе выполнения программы может содержать в разные моменты различные значения, называется _____.

2.12. Напечатается ли что-нибудь при исполнении каждого из нижеследующих операторов С? Если ничего не напечатается, ответьте «ничего». Пусть **x = 2** и **y = 3**.

a) `printf("%d", x);`

b) `printf("%d", x + x);`

c) `printf("x=");`

d) `printf("x=%d", x);`

e) `printf("%d = %d", x + y, y + x);`

f) `z = x + y;`

g) `scanf("%d%d", &x, &y);`

h) `/* printf("x + y = %d", x + y); */`

i) `printf("\n");`

2.13. Какие из следующих операторов С содержат переменные, участвующие в разрушающем считывании в ячейку?

a) `scanf("%d%d%d%d%d", &b, &c, &d, &e, &f);`

b) `p = i + j + k +7;`

c) `printf("Destructive read-in");`

d) `printf("a = 5");`

2.14. Дано уравнение $y = ax^3 + 7$; соответствует ли ему какой-нибудь из перечисленных ниже операторов С?

a) `y = a * x * x * x + 7;`

b) `y = a * x * x* (x + 7);`

c) `y = (a * x) * x* (x + 7);`

d) `y = (a * x) * x* x + 7;`

e) `y = a * (x * x* x) + 7;`

f) `y = a * x * (x* x + 7);`

2.15. Определите порядок, в котором будут выполняться операции в каждом из следующих операторов С, и определите, какое значение будет иметь **x** после завершения вычислений.

a) `x = 7 + 3 * 6 / 2 - 1;`

b) `x = 2 % 2 + 2 * 2 - 2 / 2;`

c) `x = (3 * 9 * (3 + (9 * 3 / (3))));`

2.16. Напишите программу, предлагающую пользователю ввести два числа, затем принимающую два числа, и выводящую на печать сумму, произведение, разность, частное и результат взятия по модулю этих чисел.

2.17. Напишите программу, выводящую на печать числа от одного до четырех в одну строку. Сделайте это следующими тремя способами:

a) Применив один оператор **printf** без спецификаций преобразования.

b) Применив один оператор **printf** с четырьмя спецификациями преобразования.

c) Применив четыре оператора **printf**.

2.18. Напишите программу, которая предлагает пользователю ввести два целых числа, получает эти числа и после этого выводит на печать большее из чисел со словами «**is larger**». Если же числа равны, должно печататься сообщение «**These numbers are equal**». Используйте изученный в этой главе вариант оператора **if**.

2.19. Напишите программу на C, которая принимает три различных числа с клавиатуры и находит их сумму, среднее, произведение, наименьшее и наибольшее. Используйте изученный в этой главе вариант оператора **if**. Диалог на экране должен выглядеть следующим образом:

```
Input three different integers: 13 27 34
Sum is 54
Average is 18
Product is 4914
Smallest is 13
Largest is 27
```

2.20. Напишите программу, считывающую радиус и выводящую на печать диаметр окружности, ее периметр и площадь. Используйте для величину 3.14159. Выполните каждое из этих вычислений внутри оператора **printf** и используйте спецификацию преобразования **%f**. (Замечание: в этой главе мы рассматривали только целые константы и переменные. В главе 3 мы рассмотрим числа с плавающей точкой, то есть числа, имеющие дробную десятичную часть.)

2.21. Напишите программу, печатающую прямоугольник, овал, стрелку и ромб, как показано на рисунке:



2.22. Что будет напечатано в результате выполнения данного оператора?

```
Printf("*\n**\n***\n****\n*****\n");
```

2.23. Напишите программу, которая считывает пять целых чисел, и затем определяет наибольшее и наименьшее из них. При написании пользуйтесь только теми методами программирования, которые были изучены в этой главе.

2.24. Напишите программу, считывающую целое число, а затем определяющую, четное оно или нечетное, и выводящую эту информацию на

печать. (Подсказка: воспользуйтесь операцией взятия по модулю. Четное число должно быть кратно двум. Любое кратное двум число дает в остатке ноль при делении на два.)

2.25. Напечатайте свои инициалы прописными буквами. Составьте каждую прописную букву из соответствующих ей символов, как показано на рисунке:

```
PPPPPPPPP
    P   P
    P   P
    P   P
     P P

  JJ
 J
J
 J
  JJJJJJJ


DDDDDDDDD
D        D
D        D
 D      D
  DDDDD
```

2.26. *Напишите программу, которая считывает два целых числа, после* чего определяет, кратно ли первое второму, и выводит эту информацию на печать. (Подсказка: воспользуйтесь операцией взятия по модулю.)

2.27. Изобразите модель шахматной доски, используя восемь операторов **printf**, а затем нарисуйте ту же самую картинку, используя минимально возможное количество операторов **printf**.

```
* * * * * * * *
 * * * * * * * *
* * * * * * * *
 * * * * * * * *
* * * * * * * *
 * * * * * * * *
* * * * * * * *
 * * * * * * * *
```

2.28. В чем разница между понятиями фатальная ошибка и не фатальная ошибка? Почему сообщение о фатальной ошибке можно считать более предпочтительным, чем сообщение о не фатальной ошибке?

2.29. В этом упражнении мы заглянем немного вперед. В этой главе вы узнали о целых числах типа **int**. Однако С может также оперировать со строчными и прописными буквами, и значительным количеством специальных символов. Для внутреннего представления этих символов С использует небольшие целые числа. Набор символов, используемых компьютером, с соответствующим каждому символу целым числом называется таблицей символов компьютера. Вы можете напечатать целое число, соответствующее, например, прописной **A**, выполнив следующий оператор:

```
printf("%d", A);
```

Напишите программу на C, которая печатает целые числа, соответствующие некоторым прописным и строчным буквам, цифрам и специальным символам. Определите по крайней мере числа, соответствующие следующим символам: **A B C a b c 0 1 2 $ * + /** и символу пробела.

2.30. Напишите программу, которая считывает число из пяти цифр, разделяет это число на отдельные цифры и выводит эти цифры на печать, отделяя одну от другой тремя пробелами. Например, если пользователь ввел число **42339**, программа должна напечатать.

```
4   2   3   3   9
```

2.31. Используя только материал, изложенный в данной главе, напишите программу, которая вычисляет квадраты и кубы чисел от 0 до 10 и, используя табуляцию, выведите на печать следующую таблицу значений:

```
number  square  cube
0       0       0
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
```

Структурная разработка программ



Цели

- Изучить основные методики решения задач.
- Научиться разработке алгоритмов методом нисходящего последовательного уточнения.
- Научиться использованию структур выбора if и if/else для принятия решений.
- Научиться использованию структуры повторения while для многократного исполнения операторов программы.
- Изучить методики повторения, управляемого счетчиком, и повторения, управляемого контрольным значением.
- Понять принципы структурного программирования.
- Изучить операции инкремента, декремента и присваивания.

Содержание

3.1. Введение
3.2. Алгоритмы
3.3. Псевдокод
3.4. Управляющие структуры
3.5. Структура выбора if
3.6. Структура выбора if/else
3.7. Структура повторения while
3.8. Формулирование алгоритмов: пример 1 (повторение, управляемое счетчиком)
3.9. Формулирование алгоритмов на основе нисходящего пошагового уточнения: пример 2 (повторение, управляемое контрольным значением)
3.10. Формулирование алгоритмов на основе нисходящего пошагового уточнения: пример 3 (вложенные управляющие структуры)
3.11. Операции присваивания
3.12. Операции инкремента и декремента

Резюме • *Распространенные ошибки программирования* • *Хороший стиль программирования* • *Советы по повышению эффективности* • *Общие методические замечания* • *Упражнения для самоконтроля* • *Ответы к упражнениям для самоконтроля* • *Упражнения*

3.1. Введение

Перед написанием программы для решения конкретной задачи важно составить себе полное представление о проблеме и иметь тщательно спланированный подход к ее решению. В следующих двух главах обсуждаются методы, облегчающие разработку структурированных компьютерных программ. В разделе 4.11 мы представим краткую сводку структурного программирования, которая свяжет воедино методы, разработанные здесь и в главе 4.

3.2. Алгоритмы

Решение любой задачи, связанной с вычислениями, включает в себя выполнение ряда действий в определенном порядке. Процедура решения задачи в виде

1. *действий*, которые надлежит выполнить, и

2. *порядка*, в котором эти действия должны быть выполнены,

называется *алгоритмом*. Следующий пример показывает важность правильного определения порядка, в котором должны выполняться действия.

Рассмотрим «алгоритм активного пробуждения», которому следует некий клерк для того, чтобы встать с постели и отправиться на работу:

Встать с постели.
Снять пижаму.
Принять душ.
Одеться.
Позавтракать.
Отправиться на работу.

В результате этой последовательности действий клерк приходит на работу хорошо подготовленным для принятия критических решений. Предположим, однако, что те же самые шаги выполняются в несколько ином порядке:

Встать с постели.
Снять пижаму.
Одеться.
Принять душ.
Позавтракать.
Отправиться на работу.

В этом случае наш служащий появится на работе мокрым до нитки. Задание порядка, в котором должны выполняться операторы, называется *управлением (программой)*. В этой и следующей главах мы исследуем возможности управления программами в языке С.

3.3. Псевдокод

Псевдокод — это искусственный неформальный язык, который помогает программистам разрабатывать алгоритмы. Псевдокод, который мы здесь представляем, особенно полезен для разработки алгоритмов, которые преобразуются затем в структурные программы на языке С. Псевдокод напоминает повседневный язык; он удобен и достаточно прост, хотя и не является подлинным языком программирования для компьютера.

Программы на псевдокоде на самом деле не выполняются на компьютерах. Скорее они просто помогают программисту «продумывать» программу перед попыткой написать ее на языке программирования, таком, например, как С. В этой главе мы даем несколько примеров эффективного использования псевдокода при разработке структурированных программ на языке С.

Псевдокод состоит исключительно из символов, поэтому программистам удобно вводить программы на псевдокоде в компьютер, используя для этого редактор. Компьютер может отображать на экране или печатать по требованию последнюю копию программы на псевдокоде. Тщательно подготовленная программа на псевдокоде может быть легко преобразована в соответствующую программу на языке С. Во многих случаях для этого достаточно простой замены операторов псевдокода их эквивалентами в С.

Псевдокод состоит только из операторов действия — тех операторов, которые будут выполняться после преобразования программы из псевдокода в С и ее запуска. Объявления не являются исполняемыми операторами. Они представляют собой сообщения компилятору. Например, объявление

```
int i;
```

просто сообщает компилятору о типе переменной **i** и дает ему указание зарезервировать место в памяти для этой переменной. Но это объявление не вызывает какого-либо действия при выполнении программы, как, например, ввод, вывод или вычисление. Некоторые программисты предпочитают в начале программы на псевдокоде перечислять все переменные с кратким упоминанием об их назначении. Повторяем, псевдокод является неформальным вспомогательным средством разработки программ.

3.4. Управляющие структуры

Обычно операторы в программе выполняются один за другим в порядке их записи. Это называется *последовательным выполнением*. Различные операторы языка С, которые мы скоро будем обсуждать, дают программисту возможность указать, что следующий оператор, подлежащий выполнению, может отличаться от очередного в последовательности. Это называется *передачей управления*.

В 60-е годы стало ясно, что в основе большинства трудностей, испытываемых группами разработки программного обеспечения, лежало бесконтрольное использование передачи управления. Вина была возложена на оператор **goto**, который позволяет программисту передавать управление в программе по одному из возможных адресов в очень широком диапазоне. Понятие так называемого *структурного программирования* стало почти синонимичным «устранению оператора **goto**».

Исследование Бома и Якопини[1] показало, что программирование возможно и при полном отсутствии операторов **goto**. Смена стиля программирования на «программирование без **goto**» стала эпохальным девизом для программистов. Но только в 70-е годы широкие круги профессиональных программистов начали принимать структурное программирование всерьез. Результаты оказались впечатляющими, поскольку группы разработки программного обеспечения сообщали об уменьшении времени разработки, более частой поставке систем в срок и завершении проектов в рамках бюджета. Ключом к успеху является попросту то, что программы, созданные на основе методов структурного программирования, более понятны, их проще отлаживать и модифицировать и, самое главное, более вероятно, что они написаны без ошибок.

Работа Бома и Якопини показала, что все программы могут быть написаны с использованием всего трех *управляющих структур*, а именно *последовательной структуры*, *структуры выбора* и *структуры повторения*. Последовательная структура, по существу, является встроенной в язык С. Если не указано иначе, компьютер автоматически выполняет операторы С один за другим в порядке их записи. Фрагмент *блок-схемы* на рис. 3.1 иллюстрирует последовательную структуру языка С.

Блок-схема является графическим представлением алгоритма или части алгоритма. При рисовании блок-схем используются некоторые символы специального назначения, такие как прямоугольники, ромбы, овалы и кружки; эти символы соединяются стрелками, называемыми *линиями перехода*.

---

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

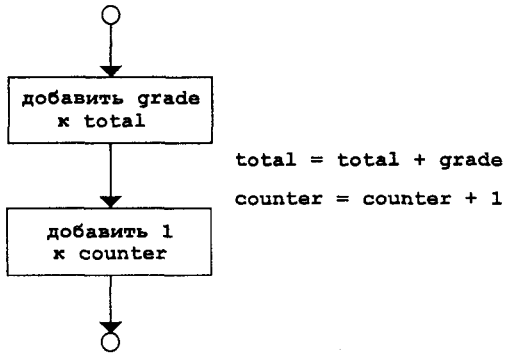


Рис. 3.1. Блок-схема последовательной структуры языка C

Подобно псевдокоду, блок-схемы полезны для разработки и представления алгоритмов, хотя большинство программистов предпочитает псевдокод. На блок-схемах ясно видно, как действуют управляющие структуры; собственно, ради этого мы и используем их в этой книге.

Рассмотрим часть блок-схемы для последовательной структуры на рис. 3.1. Для указания на любой тип действия, включая вычисление или операцию ввода/вывода, мы используем символ *прямоугольника*, называемый также *символом действия*. Линии перехода на рисунке указывают на порядок, в котором должны выполняться действия — сначала значение переменной **grade** должно быть добавлено к значению переменной **total**, а затем **1** должна быть добавлена к переменной **counter**. Язык C позволяет нам включать в последовательную структуру произвольное число действий. Как мы скоро увидим, везде, где может быть выполнено одно действие, можно поместить последовательность из нескольких действий.

При рисовании блок-схемы, представляющей *полный* алгоритм, первым символом, используемым в блок-схеме, является символ *овала*, содержащий слово «Begin» («Начало»); символ овала, содержащий слово «End» («Конец») является последним символом. При рисовании только части алгоритма, как на рис. 3.1, символы овалов опускаются и вместо них используются символы *кружков*, также называемые *символами соединения*.

Возможно, наиболее важным символом при составлении блок-схем является символ *ромба*, также называемый *символом принятия решения*, который указывает на необходимость выбора. Мы обсудим символ ромба в следующем разделе.

Язык C предоставляет программисту три типа структур выбора. В структуре выбора **if** (раздел 3.5) некоторое действие либо выполняется (выбирается), если условие истинно, либо пропускается, если это условие ложно. В структуре выбора **if/else** (раздел 3.6) некоторое действие выполняется, если условие истинно, и выполняется другое действие, если это условие ложно. В структуре выбора **switch** (обсуждаемой в главе 4) выполняется одно из набора различных действий в зависимости от значения некоторого выражения.

Структура **if** называется *структурой с единичным выбором*, поскольку в ней выбирается или игнорируется одно действие. Структура **if/else** называется *структурой с двойным выбором*, поскольку в ней выбор происходит между двумя альтернативными действиями. Структура **switch** называется *структурой со множественным выбором*, поскольку в ней выбор происходит из нескольких различных действий.

Язык С предусматривает три типа структур повторения, а именно цикл **while** (раздел 3.7) и циклы **do/while** и **for** (оба обсуждаются в главе 4).

Вот и все. В языке С имеется только семь управляющих структур: последовательная, три типа выбора и три типа повторения. Любая программа на языке С строится путем объединения такого количества управляющих структур каждого типа, которое соответствует алгоритму, реализуемому программой. Как и в случае последовательной структуры на рис. 3.1, мы увидим, что в каждой управляющей структуре имеются два символа в виде кружков, один в точке входа в управляющую структуру и один в точке выхода из нее. Эти управляющие *структуры с одним входом и одним выходом* делают написание программ довольно простым. Управляющие структуры могут присоединяться друг к другу путем соединения точки выхода одной управляющей структуры с точкой входа последующей. Это очень похоже на то, как ребенок ставит кубики один на другой, поэтому мы будем называть это *суперпозицией управляющих структур*. Мы узнаем, что существует еще только один способ их соединения — метод, называемый *вложением управляющих структур*. Таким образом, любая программа на языке С, которую нам когда-либо потребуется написать, может быть построена всего лишь из семи различных типов управляющих структур, объединенных одним из двух возможных способов.

3.5. Структура выбора if

Структура выбора используется для избрания одного из альтернативных направлений действий. Например, предположим, что проходной балл на экзамене равен 60. Оператор псевдокода

Если оценка студента больше или равна 60
Вывести на экран «Экзамен сдан»

определяет, является ли условие «оценка студента больше или равна 60» истинным или ложным. Если условие истинно, на экран выводится «Экзамен сдан» и «выполняется» следующий по порядку оператор псевдокода (не забывайте, что псевдокод не является настоящим языком программирования). Если условие ложно, вывод на экран игнорируется и выполняется следующий по порядку оператор псевдокода. Обратите внимание, что вторая строка структуры выбора записана с отступом. Использование отступов не обязательно, но мы его настоятельно рекомендуем, поскольку это помогает акцентировать внимание на существенных особенностях структуры программ. Мы будем уделять особое внимание применению соглашений об отступах на протяжении всей книги. Компилятор языка С игнорирует пробельные символы, такие как пробелы, табуляции и символы перевода строки, которые используются для введения в текст программы отступов и пустых строк. Предыдущий условный оператор псевдокода может быть написан на С как

```
if (grade >= 60)
   printf("Passed\n");
```

Обратите внимание, что код на С близко соответствует псевдокоду. Это является одним из свойств псевдокода, делающих его таким полезным инструментом разработки программ.

Хороший стиль программирования 3.1

Единообразное применение разумных соглашений об отступах значительно повышает удобочитаемость программы. Мы предлагаем использовать для отступа табуляцию фиксированного размера приблизительно в 1/4 дюйма или три пробела.

Хороший стиль программирования 3.2

Псевдокод часто используется для «продумывания» программы в процессе ее проектирования. Затем программа на псевдокоде преобразуется в программу на С.

Блок-схема на рис. 3.2 иллюстрирует структуру **if** с единичным выбором. Эта блок-схема содержит, возможно, наиболее важный из символов, используемых при составлении блок-схем — символ ромба, также называемый символом принятия решения, который указывает на необходимость принятия решения. Символ принятия решения содержит выражение, например, некоторое условие, которое может быть истинным или ложным. Символ принятия решения содержит две исходящие из него линии перехода. Одна из них показывает направление, которое должно быть выбрано, если соответствующее выражение истинно; другая показывает направление, выбираемое в том случае, если оно ложно. В главе 2 мы узнали, что решения могут приниматься на основании условий, содержащих операции отношения или равенства. На самом деле, решение может быть принято на основе любого выражения — если выражение оценивается как нулевое, то оно интерпретируется как ложное, а если оно оценивается как ненулевое, то интерпретируется как истинное.

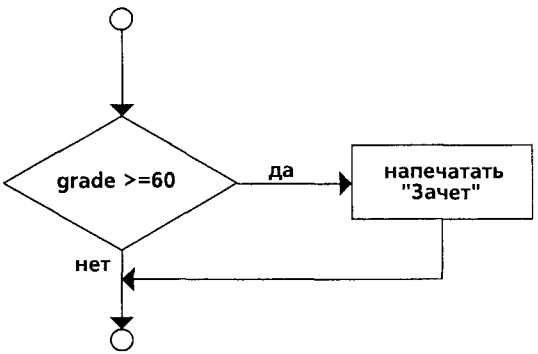


Рис. 3.2. Блок-схема структуры с единичным выбором языка С

Обратите внимание, что структура **if** является структурой с одним входом и одним выходом. Скоро мы узнаем, что блок-схемы для оставшихся управляющих структур также содержат (кроме символов в виде кружков и линий перехода) только символы прямоугольников для указания на действия, которые нужно выполнить, и символы ромбов для указания на решения, которые нужно принять. Это является моделью программирования действие/решение, на которой мы и акцентировали свое внимание.

Можно представить себе как бы семь ящиков, в каждом из которых содержатся управляющие структуры только одного из семи типов. Эти управляющие структуры пусты. В их прямоугольниках и ромбах ничего не написано.

Задачей программиста в таком случае является сборка программы из такого количества управляющих структур каждого типа, которого требует алгоритм, объединение этих управляющих структур только одним из двух возможных способов (суперпозиция или вложение) а затем заполнение символов действий и решений соответствующим алгоритму образом. Нам еще предстоит обсудить разнообразие способов написания действий и решений.

3.6. Структура выбора if/else

В структуре выбора **if** указанное действие выполняется только тогда, когда условие истинно; в противном случае действие пропускается. Структура выбора **if/else** дает программисту возможность указать, что в зависимости от того, является ли условие истинным или ложным, должны выполняться различные действия. Например, оператор псевдокода

```
Если оценка студента больше или равна 60
   Вывести на экран «Экзамен сдан»
иначе
   Вывести на экран «Экзамен не сдан»
```

выводит на экран «Экзамен сдан», если оценка студента больше или равна 60, либо выводит «Экзамен не сдан», если оценка студента меньше 60. В любом случае после вывода на экран «выполняется» следующий по порядку оператор псевдокода. Обратите внимание, что тело для ветви else также записано с отступом.

Хороший стиль программирования 3.3

Делайте отступы для обеих групп операторов структуры **if/else**.

Какое бы соглашение об отступах вы не предпочитали, оно должно последовательно применяться во всех ваших программах. Чтение программы, которая не удовлетворяет соглашению об однородности интервалов, вызывает затруднения.

Хороший стиль программирования 3.4

При наличии нескольких уровней отступов отступы для каждого уровня должны включать одно и то же количество дополнительных пробелов.

Предыдущая структура псевдокода if/else может быть написана на C как

```
if (grade >= 60)
   printf("Passed\n");
else
   printf("Failed\n");
```

Блок-схема на рис. 3.3 иллюстрирует поток управления в структуре **if/else**. Еще раз обратите внимание, что (кроме кружков и стрелок) единственными символами в блок-схеме являются прямоугольники (для действий) и ромб (для решения). Мы продолжаем акцентировать внимание на этой модели вычислений действие/решение. Снова вообразите большой ящик, содержащий такое количество пустых структур с двойным выбором, какое может по-

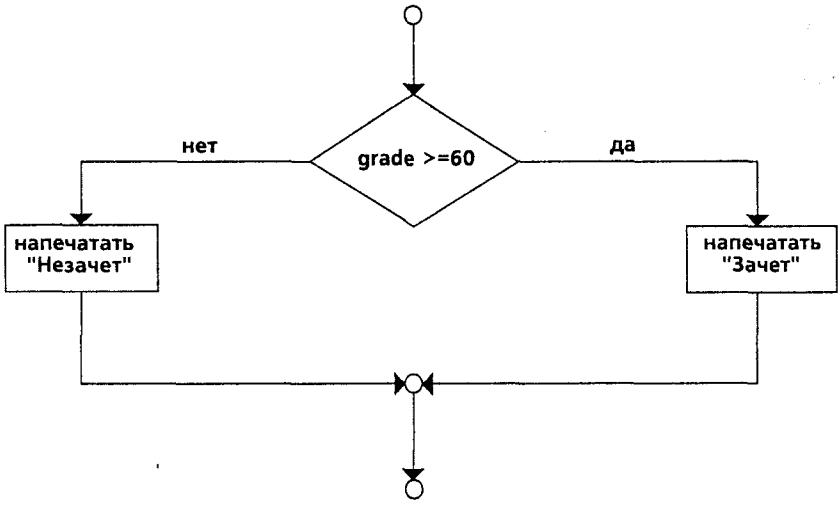


Рис. 3.3. Блок-схема структуры **if/else** с двойным выбором языка С

надобиться для написания любой программы на С. Работа программиста, опять же, состоит в компоновке этих структур выбора (путем их суперпозиции и вложения) с любыми другими управляющими структурами, которых требует алгоритм, и заполнении пустых прямоугольников и ромбов действиями и решениями, соответствующими реализуемому алгоритму.

В языке С предусмотрена условная операция (**?:**), которая тесно связана со структурой **if/else**. Условная операция является единственной тернарной (трехместной) операцией языка С, поскольку для нее требуется три операнда. Операнды вместе с условной операцией образуют условное выражение. Первый операнд является условием, второй операнд значением для всего условного выражения, если условие истинно, и третий операнд значением для всего условного выражения, если условие ложно. Например, оператор **printf**

```
printf("%s\n", grade >= 60 ? "Passed" : "Failed");
```

содержит условное выражение, значением которого является строковый литерал **«Passed» («Экзамен сдан»)**, если условие **grade >= 60** истинно, и строковый литерал **«Failed» («Экзамен не сдан»)**, если это условие ложно. Строка управления форматом функции **printf** содержит спецификацию преобразования **%s** для печати символьной строки. Таким образом, предыдущий оператор **printf** выполняется, по существу, аналогично предыдущему оператору **if/else**.

Значениями в условном выражении могут быть также действия, которые надлежит выполнить. Например, условное выражение

```
grade >= 60 ? printf("Passed\n") : printf("Failed\n");
```

читается так: «Если **оценка** больше или равна **60**, то выполнить **printf(«Passed\n»)**, иначе выполнить **printf(«Failed\n»)**». Это также сравнимо с предыдущей структурой **if/else**. Мы увидим, что в некоторых случаях условные операции могут быть использованы там, где операторы **if/else** использовать нельзя.

Вложенные структуры **if/else** служат для проверки составных условий, при этом одни структуры **if/else** помещаются внутри других структур **if/else**. Например, следующий оператор псевдокода будет печатать **A** для экзаменаци-

онных оценок, больших или равных **90**, **B** для экзаменационных оценок, больших или равных **80**, **C** для экзаменационных оценок, больших или равных **70**, **D** для экзаменационных оценок, больших или равных **60**, и **F** для всех других оценок.

```
Если оценка студента больше или равна 90
   Вывести «A»
иначе
   Если оценка студента больше или равна 80
      Вывести «B»
   иначе
      Если оценка студента больше или равна 70
         Вывести «C»
      иначе
         Если оценка студента больше или равна 60
            Вывести «D»
         иначе
            Вывести «F»
```

Этот псевдокод может быть написан на C в виде

```
if (grade >= 90)
   printf("A\n");
else
   if (grade >= 80)
      printf("B\n");
   else
      if (grade >= 70)
         printf("C\n");
      else
         if (grade >= 60)
            printf("D\n");
         else
            printf("F\n");
```

Если переменная **grade** больше или равна **90**, первые четыре условия будут истинными, однако выполняться будет только оператор **printf** после первой проверки. После выполнения этого оператора **printf** ветвь **else** «внешнего» оператора **if/else** игнорируется. Многие программисты на C предпочитают записывать предыдущую структуру **if** в виде

```
if (grade >= 90)
   printf("A\n");
else if (grade >= 80)
   printf("B\n");
else if (grade >= 70)
   printf("C\n");
else if (grade >= 60)
   printf("D\n");
else
   printf("F\n");
```

С точки зрения компилятора C обе формы являются эквивалентными. Последняя форма широко распространена, поскольку она позволяет избегать значительных отступов вправо при написании кода. При наличии таких отступов в строке часто остается мало места, что вызывает разбиение строк и ухудшает читаемость программы.

В теле структуры выбора **if** предполагается наличие только одного оператора. Чтобы включить в тело структуры **if** несколько операторов, заключите этот набор операторов в фигурные скобки ({ и }). Совокупность операторов, содержащихся внутри пары фигурных скобок, называется *составным оператором.*

Общее методическое замечание 3.1

Составной оператор может быть помещен в любое место программы, где может стоять простой оператор.

Следующий пример содержит составной оператор в ветви **else** структуры **if/else.**

```
if (grade >= 60)
   printf("Passed.\n");
else {
   printf("Failed.\n");
   printf("You must take this course again.\n");
}
```

В этом случае, если переменная **grade** меньше **60**, программа выполняет оба оператора **printf** внутри ветви **else** и выводит

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

Обратите внимание на фигурные скобки, в которые заключены оба оператора в предложении **else.** Эти фигурные скобки важны. Без фигурных скобок оператор

```
printf("You must take this course again.\n");
```

оказался бы вне тела **else**-ветви структуры **if** и выполнялся бы вне зависимости от того, меньше **60** значение **grade** или нет.

Распространенная ошибка программирования 3.1

Пропуск одной или обеих фигурных скобок, ограничивающих составной оператор.

Синтаксические ошибки вылавливаются компилятором. Логические ошибки проявляются во время выполнения. Фатальные логические ошибки являются причиной отказа в работе программы и ее преждевременного завершения. Логическая ошибка, не являющаяся фатальной, допускает дальнейшее выполнение программы, но при этом программа дает неправильные результаты.

Распространенная ошибка программирования 3.2

Помещение точки с запятой после условия в условной структуре приводит к логической ошибке в структурах **if** с одним выбором и к синтаксической ошибке в структурах **if** с двойным выбором.

Хороший стиль программирования 3.5

Некоторые программисты предпочитают вводить начальную и завершающую фигурные скобки в составных операторах до начала ввода операторов внутри фигурных скобок. Это помогает избежать пропусков одной или обеих фигурных скобок.

Общее методическое замечание 3.2

Подобно тому, как составной оператор можно помещать всюду, где может быть помещен простой оператор, так же возможно не помещать там никакого оператора вообще, то есть использовать пустой оператор. Пустой оператор представляется точкой с запятой (;) на месте, обычно занимаемом некоторым оператором.

В этом разделе мы ввели понятие составного оператора. Составной оператор может содержать объявления (как это происходит, например, в теле функции **main**). В этом случае составной оператор называется блоком. Объявления в блоке должны помещаться в начало блока перед любыми операторами действия. Мы обсудим использование блоков в главе 5. До этого времени читателю следует избегать использования блоков (конечно, это не относится к телу функции **main**).

3.7. Структура повторения while

Структура повторения позволяет программисту специфицировать многократное выполнение действия до тех пор, пока некоторое условие остается истинным. Оператор псевдокода

Пока в моем списке покупок еще остаются пункты
Сделать следующую покупку и вычеркнуть соответствующий пункт

описывает повторяющиеся действия, происходящие во время поездки за покупками. Условие «*в моем списке покупок еще остаются пункты*» может быть истинным или ложным. Если оно истинно, то выполняется действие «Сделать следующую покупку и вычеркнуть соответствующий пункт из списка». Это действие будет многократно выполняться до тех пор, пока условие будет оставаться истинным. Оператор(ы), содержащийся в структуре повторения ***while***, составляет тело этой структуры. Тело структуры ***while*** может быть простым или составным оператором.

В конце концов условие становится ложным (когда последний предмет в списке покупок приобретен и вычеркнут из списка). В этот момент повторение завершается и выполняется первый оператор псевдокода, следующий за структурой повторения.

Распространенная ошибка программирования 3.3

Отсутствие в теле структуры **while** действия, которое в конечном итоге приводит к тому, что условие в структуре **while** становится ложным. Обычно такая структура повторения никогда не будет завершена — возникает ошибка, называемая «бесконечным циклом».

Распространенная ошибка программирования 3.4

Написание ключевого слова **while** с **W** в верхнем регистре в виде **While** (не забывайте, что C является языком программирования, различающим регистр). Все зарезервированные ключевые слова языка C, такие как **while**, **if** и **else**, содержат только символы нижнего регистра.

В качестве примера реальной структуры **while** рассмотрим фрагмент программы, разработанной для нахождения первой степени 2, превосходящей 1000. Предположим, что целочисленная переменная **product** была инициализирована значением 2. После завершения выполнения следующей структуры повторения **while** в переменной **product** будет содержаться желаемый ответ:

```
product = 2;

while (product <= 1000)
   product = 2 * product;
```

Блок-схема на рис. 3.4 иллюстрирует поток управления в структуре повторения **while**. Еще раз обратите внимание, что (кроме кружков и стрелок) *блок-схема содержит только символ прямоугольника и символ ромба*. Снова вообразите себе большой ящик пустых структур **while**, которые путем суперпозиции или вложения в другие управляющие структуры могут образовывать структурную реализацию потока управления алгоритма. Пустые прямоугольники и ромбы затем заполняются соответствующими действиями и решениями. Блок-схема отчетливо демонстрирует повторение. Линия перехода, выходящая из прямоугольника, снова возвращается к блоку принятия решения, условие которого проверяется на каждом проходе цикла до тех пор, пока оно в конечном итоге не становится ложным. В этот момент происходит выход из структуры **while**, и управление переходит к следующему оператору программы.

При входе в структуру **while** значение переменной **product** равно 2. Переменная **product** многократно умножается на 2, последовательно принимая значения 4, 8, 16, 32, 64, 128, 256, 512 и 1024. Когда значение переменной **product** становится равным 1024, условие в структуре **while**, **product <= 1000**, становится ложным. Это завершает повторение, и конечное значение переменной **product** равно 1024. Выполнение программы продолжается с оператора, следующего за структурой **while**.

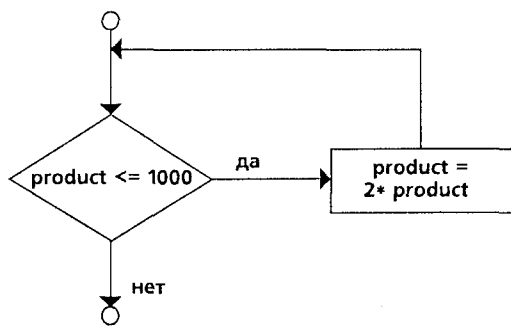


Рис. 3.4. Блок-схема структура повторения **while**

3.8. Формулирование алгоритмов: пример 1 (повторение, управляемое счетчиком)

Чтобы проиллюстрировать, как разрабатываются алгоритмы, мы решим в нескольких вариантах задачу о подсчете средней оценки в группе. Рассмотрим задачу в следующей постановке:

Группа из десяти студентов сдала экзамен. В вашем распоряжении имеются оценки (целые числа в диапазоне от 0 до 100), полученные студентами на этом экзамене. Определите среднюю оценку за экзамен для группы.

Средняя оценка равна сумме оценок, деленной на число студентов. Алгоритм для решения этой задачи на компьютере должен предусматривать ввод каждой оценки, расчет среднего и вывод результата.

Давайте воспользуемся псевдокодом и составим перечень действий, подлежащих выполнению, и определим порядок, в котором эти действия должны быть выполнены. Для ввода оценок по одной за раз мы используем *повторение, управляемое счетчиком*. Этот метод использует переменную, называемую *счетчиком*, определяющую, сколько раз должна выполняться последовательность операторов. В этом примере повторение завершается, когда значение счетчика становится больше 10. В этом разделе мы просто представим алгоритм на псевдокоде (рис. 3.5) и соответствующую программу на C (рис. 3.6). В следующем разделе мы покажем, как разрабатываются алгоритмы на псевдокоде. Повторение, управляемое счетчиком, часто называют *определенным повторением*, поскольку число повторений известно до начала выполнения цикла.

Обратите внимание на ссылки в алгоритме на итоговую сумму и счетчик. Итоговая сумма представляет собой переменную, которая используется для накопления суммы ряда значений. Счетчик — это переменная, которая используется для подсчета, в данном случае для подсчета числа введенных оценок. Переменные, используемые для хранения итоговых сумм, обычно должны инициализироваться нулем перед суммированием; иначе сумма включала бы предыдущее значение, которое хранилось в ячейке памяти итоговой суммы. Переменные-счетчики обычно инициализируются нулем или единицей в зависимости от конкретного алгоритма (мы представим примеры, иллюстрирующие каждый из вариантов). Неинициализированная переменная содержит в качестве значения «*мусор*» — значение, которое последним хранилось в зарезервированной для нее ячейке памяти.

Распространенная ошибка программирования 3.5

Если счетчик или итоговая сумма не инициализированы, результаты работы вашей программы будут, по всей вероятности, некорректными. Это является примером логической ошибки.

```
Установить итоговую сумму в ноль.
Установить счетчик оценок в единицу.

Пока счетчик оценок меньше или равен десяти
   Ввести следующую оценку
   Прибавить эту оценку к итоговой сумме
   Прибавить единицу к счетчику оценок

Присвоить средней оценке в группе значение
суммы, деленной на десять
Вывести среднюю оценку в группе
```

Рис. 3.5. Алгоритм на псевдокоде, использующий повторение, управляемое счетчиком, для решения задачи о подсчете средней оценки в группе

```
/* Программа подсчета средней оценки в группе
   с повторением, управляемым счетчиком */
#include <stdio.h>

main()
{
   int counter, grade, total, average;

   /* этап инициализации */
   total = 0;
   counter = 1;

   /* этап обработки */
   while ( counter <= 10 )  {
      printf("Enter grade: ");
      scanf("%d", &grade);
      total = total + grade;
      counter = counter + 1;
   }

   /* этап завершения */
   average = total / 10;
   printf("Class average is %d\n", average);

   return 0; /* показывает, что программа успешно завершена */
}
```

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

Рис. 3.6. Написанная на C программа подсчета средней оценки в группе с повторением, управляемым счетчиком, и пример ее выполнения

Хороший стиль программирования 3.6

Инициализируйте счетчики и итоговые суммы.

Обратите внимание, что расчет среднего в программе дал целочисленный результат. На самом деле сумма оценок в этом примере равна 817, которая, будучи поделенной на 10, должна дать 81.7, т.е. число с десятичной дробью. В следующем разделе мы увидим, как обращаться с такими числами (называемыми числами с плавающей точкой).

3.9. Формулирование алгоритмов на основе нисходящего пошагового уточнения: пример 2 (повторение, управляемое контрольным значением)

Давайте обобщим задачу о подсчете средней оценки в группе. Рассмотрим следующую задачу:

Разработать программу для подсчета средней оценки в группе, которая при каждом своем запуске будет обрабатывать произвольное число оценок.

В первом примере для подсчета средней оценки в группе число оценок (10) было известно заранее. В настоящем примере не дается никаких указаний относительно того, сколько должно быть введено оценок. Программа должна обрабатывать произвольное их число. Каким образом программа сможет определить, когда ввод оценок должен быть прекращен? Каким образом она узнает, когда должна быть вычислена и выведена на экран средняя оценка в группе?

Один из способов решения этой проблемы состоит в том, чтобы использовать для указания на «конец ввода данных» специальное значение, называемое *контрольным значением* (другие названия: сигнальное значение, фиктивное значение или флаговое значение). Пользователь вводит оценки до тех пор, пока не будут введены все «правильные» оценки. После этого пользователь вводит контрольное значение, чтобы показать, что была введена последняя оценка. Повторение, управляемое контрольным значением, часто называют неопределенным повторением, поскольку число повторений неизвестно до начала выполнения цикла.

Очевидно, что контрольное значение должно выбираться таким образом, чтобы его нельзя было спутать с допустимым входным значением. Поскольку экзаменационные оценки обычно являются неотрицательными целыми числами, приемлемым контрольным значением для этой задачи будет –1. Таким образом, при запуске программы подсчета средней оценки в группе может быть обработан поток входных значений, например, 95, 96, 75, 74, 89 и –1. После этого программа вычислит и выведет на экран среднее значение по группе для оценок 95, 96, 75, 74 и 89 (–1 является контрольным значением, поэтому оно не должно учитываться при расчете среднего).

Распространенная ошибка программирования 3.6

Выбор в качестве контрольного такого значения, которое является допустимым значением данных.

Наш подход к написанию программы для подсчета средней оценки в группе будет опираться на метод, называемый нисходящим пошаговым уточнением, и который является неотъемлемой частью разработки хорошо структурированных программ. Мы начнем с представления псевдокода для верхнего уровня нисходящего процесса:

Определить среднюю оценку в группе за экзамен

Верхний уровень представляет собой одно предложение, выражающее общее назначение программы. Как таковой, верхний уровень фактически полностью представляет программу. К сожалению, верхний уровень (как в этом случае)

редко несет в себе достаточное количество подробностей, необходимых для написания программы на С.

Итак, теперь мы начинаем процесс уточнения. Мы подразделяем верхний уровень на ряд более мелких задач и перечисляем их в том порядке, в котором они должны выполняться. Это приводит к следующему первому уточнению.

Инициализировать переменные
Ввести, просуммировать и подсчитать количество оценок
Вычислить и вывести на экран среднюю оценку для группы

Здесь имеет место последовательная структура — перечисленные шаги должны выполняться по порядку, один за другим.

Общее методическое замечание 3.3

Каждое уточнение, так же как и сам верхний уровень, представляет собой полную спецификацию алгоритма; меняется только уровень детализации.

Для перехода к следующему уровню детализации, то есть ко второму уточнению, мы вводим конкретные переменные. Нам нужна текущая сумма чисел, счетчик количества обработанных чисел, переменная для приема значения очередной вводимой оценки и переменная, в которой содержится рассчитанное среднее значение. Оператор псевдокода

Инициализировать переменные

может быть уточнен следующим образом:

Инициализировать итоговую сумму нулем
Инициализировать счетчик нулем

Обратите внимание, что нужно инициализировать только итоговую сумму и счетчик; переменные average и grade (соответственно для рассчитанного среднего значения и ввода пользователя) инициализировать не обязательно, поскольку их значения перезаписываются в результате процесса ввода с разрушением информации, обсуждаемого в главе 2. Для оператора псевдокода

Ввести, просуммировать и подсчитать количество оценок

потребуется структура повторения (т.е. цикл), в которой последовательно вводится каждая оценка. Поскольку мы не знаем заранее количества оценок, которое должно быть обработано, мы будем использовать повторение, управляемое контрольным значением. Пользователь по одной будет вводить допустимые оценки. После ввода последней допустимой оценки пользователь введет контрольное значение. Программа будет осуществлять проверку после ввода каждой оценки и завершит цикл, когда будет введено контрольное значение. Уточнением предыдущего оператора псевдокода тогда будет

Ввести первую оценку
Пока пользователь не ввел контрольного значения
    Прибавить эту оценку к текущему итогу
    Прибавить единицу к счетчику оценок
    Ввести следующую оценку (возможно, контрольное значение)

Обратите внимание, что в псевдокоде мы не используем фигурные скобки с последовательностью операторов, образующей тело структуры *while*. Мы просто выравниваем все эти операторы под *while*, чтобы показать, что все они принадлежат этой структуре *while*. Повторим, что псевдокод — это только неформальное вспомогательное средство разработки программ.

Оператор псевдокода

Вычислить и вывести на экран среднюю оценку для группы

может быть уточнен следующим образом:

```
Если счетчик не равен нулю
   Присвоить переменной average сумму, деленную на счетчик
   Вывести на экран average
иначе
   Вывести на экран «Не было ввода оценок»
```

Обратите внимание, что мы действуем здесь достаточно аккуратно и проверяем возможность деления на ноль — фатальной ошибки, которая, оставшись невыявленной, способна привести к отказу в работе программы (часто называемому «крахом»). Полностью второе уточнение показано на рис. 3.7.

Распространенная ошибка программирования 3.7

Попытка деления на ноль, являющаяся причиной фатальной ошибки.

```
Инициализировать итоговую сумму нулем
Инициализировать счетчик нулем

Ввести первую оценку
   Пока пользователь не ввел контрольного значения
   Прибавить эту оценку к текущему итогу
   Прибавить единицу к счетчику оценок
   Ввести следующую оценку (возможно, контрольное значение)

Если счетчик не равен нулю
   Присвоить переменной average сумму, деленную на счетчик
   Вывести на экран average
иначе
   Вывести на экран «Не было ввода оценок»
```

Рис. 3.7. Алгоритм на псевдокоде, использующий повторение, управляемое контрольным значением, для решения задачи о подсчете средней оценки в группе

Хороший стиль программирования 3.7

При выполнении деления на выражение, значение которого может быть нулем, явно осуществляйте проверку этого случая и обрабатывайте его соответствующим образом (например, выводите сообщение об ошибке), и не допускайте возникновения фатальной ошибки.

На рис. 3.5 и 3.7 мы включаем в псевдокод несколько пустых строк для повышения его удобочитаемости. Фактически пустые строки разделяют эти программы на различные этапы их выполнения.

Общее методическое замечание 3.4

Многие программы логически могут быть разделены на три этапа: этап инициализации, на котором происходит инициализация переменных программы; этап обработки, на котором происходит ввод данных и соответствующая настройка переменных программы; и этап завершения работы программы, на котором происходит вычисление и вывод окончательных результатов.

Алгоритм на псевдокоде, приведенный на рис. 3.7, решает задачу о подсчете средней оценки в группе в более общей постановке. Этот алгоритм был разработан после всего лишь двух этапов уточнения. Иногда требуется большее количество этапов.

Общее методическое замечание 3.5

Программист завершает процесс нисходящего пошагового уточнения в тот момент, когда алгоритм на псевдокоде специфицирован достаточно подробно для того, чтобы программист мог преобразовать псевдокод в программу на С. Реализация программы на С в этом случае обычно не вызывает затруднений.

На рис. 3.8 показана программа на С и пример ее выполнения. Хотя вводятся только целочисленные оценки, при расчете среднего может получиться дробное число. Типом **int** такое число представлено быть не может. В программе применяется тип данных **float** для обработки чисел с десятичной дробью (называемых *числами с плавающей точкой*) и специальная операция, называемая *операцией приведения типа*, для управления расчетом среднего значения. Более подробно эти понятия объясняются после представления программы.

Обратите внимание на составной оператор в цикле **while** на рис. 3.8. Повторим, что фигурные скобки необходимы для того, чтобы выполнялись все четыре оператора внутри цикла. Без фигурных скобок последние три оператора в теле цикла вышли бы за его пределы, приводя к неправильной интерпретации этого кода:

```
while (grade != -1)
   total = total + grade;
counter = counter +1;
printf("Enter grade, -1 to end: ");
scanf("%d", &grade);
```

Получается бесконечный цикл, если пользователь не вводит –1 в качестве первой оценки.

Хороший стиль программирования 3.8

В цикле, управляемом контрольным значением, подсказки для запроса ввода данных должны явно напоминать пользователю, чему равно это значение.

Средние величины не всегда выражаются целочисленными значениями. Часто среднее является значением типа 7.2 или –93.5, которое содержит

дробную часть. Эти значения называются числами с плавающей точкой и представляются типом данных **float**. Переменная **average** объявлена как **float**, чтобы не потерять дробную часть результата нашего вычисления.

```
/* Программа подсчета средней оценки в группе
    с повторением, управляемым контрольным значением */
#include <stdio.h>

main ()
{
   float average;                    /* новый тип данных */
   int counter, grade, total;

   /* этап инициализации */
   total = 0;
   counter = 0;

   /* этап обработки */
   printf("Enter grade, -1 to end: ");
   scanf("%d", &grade);

   while ( grade != -1) {
      total = total + grade;
      counter = counter + 1;
      printf("Enter grade, -1 to end: ");
      scanf("%d", &grade);
   }

   /* этап завершения */
   if (counter != 0 ) {
      average = (float) total / counter;
      printf("Class average is %.2f", average);
   }
   else
      printf("No grades were entered\n");

   return 0;  /* показывает, что программа успешно завершена */
}
```

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

Рис. 3.8. Написанная на C программа подсчета средней оценки в группе с повторением, управляемым контрольным значением, и пример ее выполнения

Однако результат вычисления **total / counter** является целым числом, поскольку обе переменных **total** и **counter** являются целочисленными. Деление двух целых чисел приводит к *целочисленному делению*, при котором дробная часть результата теряется (т.е. происходит его *усечение*). Поскольку сначала выполняется деление, дробная часть теряется до присваивания результата переменной **average**. Для деления с плавающей точкой целочисленных значений мы должны создать для них временные значения, которые являются числами с плавающей точкой.

В языке С для решения этой задачи предусмотрена одноместная *операция приведения типа*. Оператор

```
average = (float) total / counter;
```

включает в себя операцию приведения типа (**float**), которая создает для своего операнда **total** временную копию с плавающей точкой. Использование операции приведения типа подобным образом называется *явным преобразованием*. Значение, хранимое в переменной **total**, по-прежнему является целым числом. Вычисление теперь состоит в делении значения с плавающей точкой (временная «копия» **total** типа **float**) на целочисленное значение, хранимое в переменной **counter**. Компилятор С умеет оценивать лишь выражения с идентичными типами данных операндов. Чтобы гарантировать совпадение типа операндов, компилятор выполняет над выбранными операндами так называемую операцию *возведения типа* (или *неявное преобразование*). Например, в выражении, содержащем типы данных **int** и **float**, согласно стандарту ANSI, для операндов типа **int** делаются копии, которые возводятся до типа **float**. В нашем примере после создания копии переменной **counter** и ее возведения до типа **float** производится вычисление, и результат деления с плавающей точкой присваивается переменной **average**. Стандартом ANSI предусматривается набор правил возведения для операндов различных типов. В главе 5 представлено обсуждение всех стандартных типов данных и порядка их возведения.

Операции приведения типа существуют для любого типа данных. Операция приведения типа образуется путем помещения имени типа данных в круглые скобки. Операция приведения типа является *одноместной операцией*, т.е. операцией, принимающей только один операнд. В главе 2 мы изучали двухместные арифметические операции. В С также поддерживаются *одноместные варианты* операций плюс (+) и минус (-), поэтому программист может написать выражения вроде **-7** или **+5**. Операции приведения типа ассоциируются справа налево и имеют тот же приоритет, что и другие одноместные (унарные) операции, как, например, унарный + и унарный -. Этот приоритет на один уровень выше приоритета мультипликативных операций *, / и % и на один порядок ниже приоритета круглых скобок.

В программе на рис. 3.8 для вывода значения переменной **average** используется спецификатор преобразования **%.2f** функции **printf**. Символ **f** указывает, что будет выведено значение с плавающей точкой. Символ **.2** представляет собой точность, с которой будет отображено это значение. Она устанавливает, что значение будет отображено с 2 десятичными знаками справа от десятичной точки. Если используется спецификатор преобразования **%f** (без указания точности), то по умолчанию используется точность 6 — как если бы был указан спецификатор преобразования **%.6f**. Когда значения с плавающей точкой выводятся с указанием точности, выводимое значение округляется до

заданного числа десятичных знаков. Значение в памяти остается неизменным. При выполнении следующих операторов выводятся значения 3.45 и 3.4.

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

Распространенная ошибка программирования 3.8

Указание точности для спецификации преобразования в строке управления форматом функции **scanf** является ошибочным. Точность используется только в спецификациях преобразования функции **printf**.

Распространенная ошибка программирования 3.9

Предположение, что числа с плавающей точкой будут представлены в памяти компьютера с абсолютной точностью, может привести к неправильным результатам. На большинстве компьютеров числа с плавающей точкой представляются только приблизительно.

Хороший стиль программирования 3.9

Не проверяйте на равенство значения с плавающей точкой.

Несмотря на то, что числа с плавающей точкой не всегда «на 100% точны», они имеют многочисленные приложения. Например, когда мы говорим о «нормальной» температуре тела в 98.6 градусов (по Фаренгейту), нам не нужна точность, представляемая большим числом знаков. Когда термометр показывает нам температуру в 98.6 градусов, на самом деле она может быть равна 98.5999473210643. Суть в том, что принятие этого числа равным просто 98.6 прекрасно подходит для большинства приложений. Позже мы обсудим этот вопрос более подробно.

Другой причиной, по которой возникают числа с плавающей точкой, является операция деления. Когда мы делим 10 на 3, результатом является число 3.3333333... с бесконечной повторяющейся последовательностью 3. Компьютер выделяет лишь ограниченную область памяти для его хранения, поэтому очевидно, что значение с плавающей точкой может быть только приближенным.

3.10. Формулирование алгоритмов на основе нисходящего пошагового уточнения: пример 3 (вложенные управляющие структуры)

Давайте найдем полное решение еще одной задачи. Мы еще раз сформулируем алгоритм, используя для этого псевдокод и процесс нисходящего пошагового уточнения, и напишем соответствующую программу на C. Мы уже видели, что управляющие структуры могут (последовательно) помещаться одна на другую, подобно тому, как ребенок укладывает кубики. В этом примере мы увидим единственный отличный от предыдущего способ, которым в C могут соединяться управляющие структуры, а именно *вложение* одной управляющей структуры в другую.

Рассмотрим следующую постановку задачи:

Колледж предлагает платный курс для подготовки студентов к сдаче государственного экзамена на получение диплома брокера по недвижимости. В прошлом году несколько студентов из прослушавших этот курс сдавали экзамен на получение диплома. Естественно, что колледж хотел бы знать, насколько успешно его студенты сдали этот экзамен. Вас попросили написать программу для обобщения результатов экзамена. Вам дали список этих 10 студентов. Напротив фамилии каждого студента проставлена 1, если студент сдал экзамен, и 2, если он экзамен не сдал.

Ваша программа должна анализировать результаты экзамена следующим образом:

1. Ввести результат каждого экзамена (т.е. 1 или 2). При каждом очередном запросе результата экзамена отображать на экране сообщение «Введите результат».
2. Подсчитать количество результатов каждого типа.
3. Отобразить краткую сводку результатов экзамена, указав количество студентов, сдавших экзамен, и количество студентов, его не сдавших.
4. Если экзамен сдали более 8 студентов, вывести сообщение «Повысить плату за курс».

После тщательного изучения постановки задачи мы можем сделать следующие наблюдения:

1. Программа должна обработать 10 результатов экзамена. Будет использован цикл, управляемый счетчиком.
2. Каждый результат экзамена представляет собой число, 1 или 2. Всякий раз, когда программа считывает результат экзамена, она должна определять, является ли это число 1 или 2. В нашем алгоритме мы проверяем на 1. Если число не равно 1, мы предполагаем, что это 2. (В упражнении в конце этой главы рассматриваются последствия такого предположения.)
3. Используются два счетчика — один для подсчета числа студентов, сдавших экзамен, и другой для подсчета числа студентов, его не сдавших.
4. После того, как программа обработает все результаты, она должна решить, сдали ли экзамен более 8 студентов.

Давайте перейдем к нисходящему пошаговому уточнению алгоритма. Мы начинаем с представления на псевдокоде верхнего уровня:

Проанализировать результаты экзамена и решить, должна ли быть повышена плата за курс

Повторим — важно акцентировать внимание на том, что верхний уровень полностью представляет программу, однако, по всей вероятности, потребуется несколько уточнений, прежде чем псевдокод можно будет естественным образом преобразовать в программу на C. Нашим первым уточнением является

Инициализировать переменные
Ввести десять экзаменационных оценок и подсчитать число сданных и несданных экзаменов
Вывести краткую сводку результатов экзамена и решить, должна ли быть повышена плата за курс

Хотя здесь мы также имеем полное представление всей программы, необходимо дальнейшее уточнение. Теперь мы введем конкретные переменные. Нам потребуются счетчики для регистрации количества сданных и несданных экзаменов, счетчик для управления выполнением цикла и переменная для хранения пользовательского ввода. Оператор псевдокода

Инициализировать переменные

может быть уточнен следующим образом:

Инициализировать нулем количество сданных экзаменов
Инициализировать нулем количество несданных экзаменов
Инициализировать счетчик студентов единицей

Обратите внимание, что инициализируются только счетчики и итоговые суммы. Оператор псевдокода

Ввести десять экзаменационных оценок и подсчитать число сданных и несданных экзаменов

требует цикла для последовательного ввода результатов каждого экзамена. Здесь известно заранее, что имеется в точности десять результатов экзамена, следовательно, подойдет цикл, управляемый счетчиком. Структура с двойным выбором внутри цикла (т.е. вложенная в цикл) будет определять для каждого экзамена, является ли он в результате сданным или несданным, и исходя из этого увеличивать соответствующий счетчик. Уточнением предыдущего оператора псевдокода тогда будет

```
Пока счетчик студентов меньше или равен десяти
   Ввести следующий результат экзамена

   Если студент сдал экзамен
      Прибавить единицу к счетчику сданных экзаменов
   иначе
      Прибавить единицу к счетчику несданных экзаменов

   Прибавить единицу к счетчику студентов
```

Обратите внимание на вставку пустых строк для выделения управляющей структуры **if/else**, делающих программу более удобочитаемой. Оператор псевдокода

Вывести краткую сводку результатов экзамена и решить, должна ли быть повышена плата за курс

может быть уточнен следующим образом:

```
Вывести количество сданных экзаменов
Вывести количество несданных экзаменов
Если экзамен сдали более восьми студентов
   Вывести «Повысить плату за курс»
```

Полностью второе уточнение показано на рис. 3.9. Обратите внимание, что и в данном случае для выделения структуры *while* вставлены пустые строки, что повышает удобочитаемость программы.

```
Инициализировать нулем количество сданных экзаменов
Инициализировать нулем количество несданных экзаменов
Инициализируйте счетчик студентов единицей

Пока счетчик студентов меньше или равен десяти
   Ввести следующий результат экзамена

   Если студент сдал экзамен
      Прибавить единицу к счетчику сданных экзаменов
   иначе
      Прибавить единицу к счетчику несданных экзаменов

   Прибавить единицу к счетчику студентов

Вывести количество сданных экзаменов
Вывести количество несданных экзаменов
Если экзамен сдали более восьми студентов
   Вывести «Повысить плату за курс»
```

Рис. 3.9. Псевдокод для задачи о результатах экзаменов

Теперь псевдокод является достаточно детализированным для преобразования его в код С. Программа на С и два примера ее выполнения показаны на рис. 3.10. Обратите внимание, что мы воспользовались свойством языка С, позволяющим инициализацию объединять с объявлениями. Подобного рода инициализация происходит во время компиляции.

Совет по повышению эффективности 3.1

Инициализация переменных при их объявлении уменьшает время выполнения программы.

Общее методическое замечание 3.6

Опыт показывает, что наиболее трудной частью решения задачи на компьютере является разработка алгоритма ее решения. Как только соответствующий алгоритм построен, процесс написания работающей программы на С обычно не вызывает затруднений.

Общее методическое замечание 3.7

Многие программисты пишут программы, никогда не пользуясь никакими инструментами разработки программ типа псевдокода. Они полагают, что поскольку их конечной целью является решение задачи на компьютере, написание псевдокода только задерживает производство конечного продукта.

```
/* Анализ результатов экзамена */
#include <stdio.h>

main()
{
   /* инициализация переменных при их объявлении */
   int passes = 0, failures = 0, student = 1, result;

   /* обрабатывает 10 студентов; цикл, управляемый счетчиком */
   while (student <= 10) {
      printf("Enter result (1=pass, 2=fail): ");
      scanf("%d", &result);

      if (result == 1)        /* if/else вложенная в while */
         passes = passes + 1;
      else
         failures = failures + 1;

      student = student + 1;
   }

   printf("Passed %d\n", passes);
   printf("Failed %d\n", failures);

   if (passes > 8)
      printf("Raise tuition\n");

   return 0;    /* успешное завершение */
}
```

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

Enter result (1=pass, 2=fail): 1
Enter result (1=pass, 2=fail): 1
Enter result (1=pass, 2=fail): 1
Enter result (1=pass, 2=fail): 2
Enter result (1=pass, 2=fail): 1
Enter result (1=pass, 2=fail): 1
Enter result (1=pass, 2=fail): 1
Enter result (1=pass, 2=fail): 1
Enter result (1=pass, 2=fail): 1
Passed 9
Failed 1
Raise tuition
```

Рис. 3.10. Программа на C и пример ее выполнения для задачи о результатах экзаменов

3.11. Операции присваивания

В языке C предусмотрено несколько операций присваивания для более короткой записи выражений. Например, оператор

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

может быть сокращен с помощью операции сложения с присваиванием += до

```
c += 3;
```

Операция += прибавляет значение выражения справа от операции к значению переменной слева от нее и сохраняет результат в переменной слева от операции. Любой оператор вида

переменная = переменная операция выражение;

где операция является одной из двухместных операций +, -, *, / или % (других, которые мы обсудим в главе 10), может быть записан в виде

переменная операция= выражение;

Таким образом, присваивание **c += 3** прибавляет **3** к значению переменной **c**. На рис. 3.11 показаны арифметические операции присваивания, примеры выражений, в которых они используются, и соответствующие пояснения.

Операция присваивания	Пример выражения	Пояснение	Присваивает
Предположим: `int c = 3, d = 5, e = 4, f = 6, g = 12;`			
+=	`c += 7`	`c = c + 7`	10 переменной c
-=	`d -= 4`	`d = d - 4`	1 переменной d
*=	`e *= 5`	`e = e * 5`	20 переменной e
/=	`f /= 3`	`f = f / 3`	2 переменной f
%=	`g %= 9`	`g = g % 9`	3 переменной g

Рис. 3.11. Арифметические операции присваивания

Совет по повышению эффективности 3.2

*Выражение с операцией присваивания (например, **c += 3**) компилируется быстрее, чем эквивалентное раскрытое выражение (**c = c + 3**), поскольку переменная **c** в первом выражении оценивается только один раз, в то время как во втором выражении она оценивается дважды.*

Совет по повышению эффективности 3.3

Многие из советов по повышению эффективности, которые мы даем в этой книге, приводят к незначительным на первый взгляд улучшениям, поэтому у читателя может возникнуть искушение игнорировать их. Дело в том, что заставить программу выполняться значительно быстрее может только совокупный эффект от всех этих мер по повышению эффективности. Кроме того, значительное улучшение достигается при внесении вроде бы незначительного усовершенствования в цикл, который может повторяться большое количество раз.

3.12. Операции инкремента и декремента

В языке C предусмотрены также унарная *операция инкремента* ++ и унарная *операция декремента* --, краткая информация о которых приведена на рис. 3.12. При увеличении переменной **c** на 1 вместо выражений **c = c + 1** или **c += 1** может использоваться операция инкремента ++. Если операции инкремента или декремента помещаются перед переменной, они называются соответственно *преинкрементными* или *предекрементными* операциями. Если операции инкремента или декремента помещаются после переменной, они называются соответственно *постинкрементными* или *постдекрементными* операциями. Операция преинкремента (предекремента) над переменной вызывает увеличение (уменьшение) этой переменной на 1, затем новое значение переменной используется в выражении, в котором она появляется. Операция постинкремента (постдекремента) над переменной вызывает использование текущего значения этой переменной в выражении, вкотором она появляется, затем значение переменной увеличивается (уменьшается) на 1.

Операция	Пример выражения	Пояснение
++	++a	Увеличивает a на 1, затем использует новое значение a в выражении, в которое входит a.
++	a++	Использует текущее значение a в выражении, в которое входит a, затем увеличивает a на 1.
--	--b	Уменьшает b на 1, затем использует новое значение b в выражении, в которое входит b.
--	b--	Использует текущее значение b в выражении, в которое входит b, затем уменьшает b на 1.

Рис. 3.12. Операции инкремента и декремента

В программе, приведенной на рис. 3.13, демонстрируется различие между преинкрементной и постинкрементной версиями операции ++. Постинкрементирование переменной c вызывает ее увеличение после ее использования в функции **printf**. Преинкрементирование переменной c вызывает ее увеличение до ее использования в функции **printf**.

Программа отображает значение переменной c до и после использования операции ++. Операция декремента (--) действует аналогично.

Хороший стиль программирования 3.10

Унарные операции следует помещать непосредственно рядом с их операндами без разделительных пробелов.

Три оператора присваивания на рис. 3.10

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

могут быть записаны более кратко с использованием операций присваивания в виде

```
/* Операции преинкремента и постинкремента */
#include <stdio.h>

main()
{
   int c;

   c = 5;
   printf("%d\n", c);
   printf("%d\n", c++);       /* постинкремент */
   printf("%d\n\n", c);

   c = 5;
   printf("%d\n", c);
   printf("%d\n", ++c);       /* преинкремент */
   printf("%d\n", c);

   return 0;     /* успешное завершение */
}
```

```
5
5
6

5
6
6
```

Рис. 3.13. Демонстрация различия между операциями преинкремента и постинкремента

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

с использованием операций преинкремента в виде

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

или с операциями постинкремента

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

Здесь важно обратить внимание на то, что при инкрементировании или декрементировании переменной в операторе, в который входит только эта переменная, преинкрементная и постинкрементная формы дают один и тот же эффект. Только при появлении переменной в контексте большего выражения преинкрементная и постинкрементная формы имеют различный смысл (это же верно и для операций предекремента и постдекремента).

В качестве операнда операций инкремента или декремента может быть использовано только простое имя переменной.

Распространенная ошибка программирования 3.10

Попытка использовать операцию инкремента или декремента с выражением, отличным от простого имени переменной, вроде **++(x + 1)**, является синтаксической ошибкой.

Хороший стиль программирования 3.11

Стандартом ANSI в общем случае не специфицируется порядок, в котором будут оцениваться операнды той или иной операции (хотя в главе 4 мы увидим, что для некоторых операций из этого правила существуют исключения). Поэтому программист должен избегать команд с операциями инкремента или декремента, в которых некоторая переменная подвергается их воздействию более одного раза.

В таблице на рис. 3.14 показаны приоритет и ассоциативность представленных до сих пор операций. Операции показаны сверху вниз в порядке уменьшения их приоритета. Во втором столбце описывается ассоциативность операций для каждого уровня приоритетов. Обратите внимание, что условная операция (?:), унарные операции инкремента (++), декремента (--), плюс (+), минус (-) и приведения типа, и операции присваивания =, +=, -=, * =, /= и %= ассоциированы справа налево. В третьем столбце указаны названия различных групп операций. Все другие операции на рис. 3.14 ассоциированы слева направо.

Операции	Ассоциативность	Тип
()	слева направо	круглые скобки
++ -- + - (тип)	справа налево	унарные
* / %	слева направо	мультипликативные
+ -	слева направо	аддитивные
< <= > >=	слева направо	отношения
== !=	слева направо	равенства
?:	справа налево	условные
= += -= *= /= %=	справа налево	присваивания

Рис. 3.14. Приоритет операций, упомянутых в книге до настоящего момента

Резюме

- Решение любой задачи, связанной с вычислениями, включает в себя выполнение ряда действий в определенном порядке. Процедура решения задачи в виде действий, которые надлежит выполнить, и порядка, в котором эти действия должны быть выполнены, называется алгоритмом.
- Задание порядка, в котором в компьютерной программе должны выполняться операторы, называется программным управлением.
- Псевдокод — это искусственный неформальный язык, который помогает программистам разрабатывать алгоритмы. Он напоминает повседневный язык. Программы на псевдокоде на самом деле не выполняются на компьютерах. Скорее они просто помогают программисту «продумывать» программу перед попыткой написать ее на языке программирования, таком, например, как C.

- Псевдокод состоит исключительно из символов, поэтому программисты могут вводить программы на псевдокоде в компьютер, редактировать и сохранять их.
- Псевдокод состоит только из исполняемых операторов. Объявления представляют собой сообщения компилятору, которые передают ему атрибуты переменных и дают ему указание зарезервировать для них место в памяти.
- Структура выбора используется для избрания одного из альтернативных направлений действия.
- В структуре выбора **if** указанное действие выполняется только тогда, когда условие истинно.
- Структура выбора **if/else** указывает, что в зависимости от истинности или ложности условия должны выполняться различные действия.
- Во вложенной структуре выбора **if/else** может проверяться несколько различных случаев. Если истинным оказывается более одного условия, будут выполнены только операторы после первого истинного условия.
- Всякий раз, когда должно быть выполнено более одного оператора в том месте программы, где обычно ожидается появление только одного оператора, эти операторы должны быть заключены в фигурные скобки, образующие составной оператор. Составной оператор может быть помещен в любое место программы, где может стоять простой оператор.
- Признаком пустого оператора, показывающего, что не должно выполняться никакое действие, служит точка с запятой (;) на месте, обычно занимаемом некоторым оператором.
- Структура повторения требует многократного выполнения действия до тех пор, пока некоторое условие остается истинным.
- Формат структуры повторения **while**:

```
while (условие)
   оператор
```

  Оператор (а также составной оператор или блок), содержащийся в структуре повторения **while**, составляет тело этой структуры.
- Обычно некоторое действие, выполняемое в теле структуры **while**, должно в конечном итоге приводить к тому, что условие становится ложным. В противном случае цикл никогда не будет завершен — возникает ошибка, называемая «бесконечным циклом».
- В цикле, управляемом счетчиком, для определения момента завершения цикла используется некоторая переменная — счетчик проходов цикла.
- Итоговая сумма представляет собой переменную, в которой накапливается сумма последовательности чисел. Итоговые суммы обычно должны быть инициализированы нулем до выполнения цикла.
- Блок-схема является графическим представлением алгоритма. При рисовании блок-схем используются некоторые специальные символы, такие как овалы, прямоугольники, ромбы и кружки, соединяемые стрелками, называемыми линиями перехода. Символы указывают на

действия, которые подлежат выполнению. Линии перехода указывают на порядок, в котором должны выполняться действия.

- Символ овала, называемый также конечным символом, указывает на начало и конец всякого алгоритма.
- Символ прямоугольника, называемый также символом действия, указывает на вычисления любого типа или операцию ввода/вывода. Символы прямоугольников соответствуют действиям, которые обычно выполняются с помощью операторов присваивания, или операторам ввода/вывода, которые реализуются при посредстве стандартных библиотечных функций, подобных **printf** и **scanf**.
- Символ ромба, также называемый символом принятия решения, указывает на необходимость выбора. Символ принятия решения содержит выражение, которое может быть истинным или ложным. Из него исходят две линии перехода. Одна линия перехода показывает направление, которое должно быть выбрано, если условие истинно; другая показывает направление, выбираемое в том случае, когда условие ложно.
- Значение, которое содержит дробную часть, называется числом с плавающей точкой и представляется типом данных **float**.
- Деление двух целых чисел приводит к целочисленному делению, при котором дробная часть вычисления теряется (т.е. происходит усечение числа).
- В языке С предусмотрена одноместная операция приведения типа (**float**), которая создает для своего операнда временную копию с плавающей точкой. Использование операции приведения типа подобным образом называется *явным преобразованием*. Операции приведения типа существуют для любого типа данных.
- Компилятор С умеет оценивать лишь выражения с идентичными типами данных операндов. Чтобы гарантировать совпадение типа операндов, компилятор выполняет над выбранными операндами так называемую операцию возведения типа (или неявное преобразование). Стандартом ANSI специфицируется, что для операндов типа **int** делаются копии, которые возводятся до типа **float**. Стандарт ANSI предусматривает набор правил возведения для операндов различных типов.
- Значения с плавающей точкой выводятся с определенным числом знаков после десятичной точки; оно задается спецификатором преобразования **%f** в операторе **printf**. Значение **3.456**, выведенное со спецификатором преобразования **%.2f**, отображается на экране как **3.46**. Если используется спецификатор преобразования **%f** (без указания точности), то по умолчанию принимается точность, равная 6.
- В языке С предусмотрены разнообразные операции присваивания, помогающие записывать более коротко некоторые распространенные типы арифметических выражений присваивания. Эти операции таковы: **+=**, **-=**, ***=**, **/=** и **%=**. Вообще любой оператор вида

```
переменная = переменная операция выражение;
```

где операция является одной из двухместных операций +, -, *, / или % может быть записан в виде

переменная операция= выражение;

- В языке С предусмотрена операция инкремента ++ и операция декремента -- для увеличения или уменьшения значения переменной на 1. Эти операции могут применяться к переменным в префиксной или постфиксной формах. Если операция используется в префиксной форме, то переменная сначала увеличивается или уменьшается на 1 и затем участвует в оценке выражения. Если операция применяется в постфиксной форме, то переменная участвует в оценке и затем увеличивается или уменьшается на 1.

Терминология

float
алгоритм
арифметические операции присваивания: +=, -=, *=, /= и %=
бесконечный цикл
блок
блок-схема
вложенные структуры **if/else**
вложенные управляющие структуры
возведение
второе уточнение
выбор
действие
деление на ноль
инициализация
исключение **goto**
итоговая сумма
«конец ввода данных»
конечный символ
контрольное значение
крах программы
линии перехода
логическая ошибка
мультипликативные операции
«мусор»
неопределенное повторение
нефатальная ошибка
неявное преобразование
нисходящее пошаговое уточнение
округление
оператор goto
операция декремента (—)
операция инкремента (++)
операция постдекремента
операция постинкремента
операция предекремента
операция преинкремента
операция приведения
определенное повторение
отказ программы
первое уточнение
передача управления
повторение
повторение, управляемое счетчиком
порядок действий
последовательная структура
последовательное выполнение
пошаговое уточнение
пробельные символы
псевдокод
пустой оператор (;)
решение
сигнальное значение
символ действия
символ завершения
символ овала
символ прямоугольника
символ решения
символ ромба
символ стрелки
символы блок-схемы
синтаксическая ошибка
составной оператор
структура выбора **if**
структура выбора **if/else**
структура повторения **while**
структура с двойным выбором
структура с единичным выбором
структура со множественным выбором
структурное программирование
структуры выбора
структуры повторения
суперпозиция управляющих структур
счетчик
тело цикла
тернарная (трехместная) операция
точность
точность по умолчанию
управление программой
управляющая структура
усечение

управляющие структуры с одним входом / одним выходом
условие завершения
условная операция (?:)
фаза завершения
фаза инициализации
фаза обработки
фатальная ошибка
фиктивное значение
флаговое значение
целочисленное деление
цикл
число с плавающей точкой
шаг
явное преобразование

Распространенные ошибки программирования

3.1. Пропуск одной или обеих фигурных скобок, ограничивающих составной оператор.

3.2. Помещение точки с запятой после условия в условной структуре приводит к логической ошибке в структурах **if** с одним выбором и к синтаксической ошибке в структурах **if** с двойным выбором.

3.3. Отсутствие в теле структуры **while** действия, которое в конечном итоге приводит к тому, что условие в структуре **while** становится ложным. Обычно такая структура повторения никогда не будет завершена — возникает ошибка, называемая «бесконечным циклом».

3.4. Написание ключевого слова **while** с **W** в верхнем регистре в виде **While** (не забывайте, что C является языком программирования, различающим регистр). Все зарезервированные ключевые слова языка C, такие как **while**, **if** и **else**, содержат только символы нижнего регистра.

3.5. Если счетчик или итоговая сумма не инициализированы, результаты работы вашей программы будут, по всей вероятности, некорректными. Это является примером логической ошибки.

3.6. Выбор в качестве контрольного такого значения, которое является допустимым значением данных.

3.7. Попытка деления на ноль, являющаяся причиной фатальной ошибки.

3.8. Указание точности для спецификации преобразования в строке управления форматом функции **scanf** является ошибочным. Точность используется только в спецификациях преобразования функции **printf**.

3.9. Предположение, что числа с плавающей точкой будут представлены в памяти компьютера с абсолютной точностью, может привести к неправильным результатам. На большинстве компьютеров числа с плавающей точкой представляются только приблизительно.

3.10. Попытка использовать операцию инкремента или декремента с выражением, отличным от простого имени переменной, вроде **++(x + 1)**, является синтаксической ошибкой.

Хороший стиль программирования

3.1. Единообразное применение разумных соглашений об отступах значительно повышает удобочитаемость программы. Мы предлагаем ис-

пользовать для отступа табуляцию фиксированного размера приблизительно в 1/4 дюйма или три пробела.

3.2. Псевдокод часто используется для «продумывания» программы в процессе ее проектирования. Затем программа на псевдокоде преобразуется в программу на C.

3.3. Делайте отступы для обеих групп операторов структуры **if/else**.

3.4. При наличии нескольких уровней отступов отступы для каждого уровня должны содержать одно и то же количество дополнительных пробелов.

3.5. Некоторые программисты предпочитают вводить начальную и завершающую фигурные скобки в составных операторах до начала ввода операторов внутри фигурных скобок. Это помогает избежать пропусков одной или обеих фигурных скобок.

3.6. Инициализируйте счетчики и итоговые суммы.

3.7. При выполнении деления на выражение, значение которого может быть нулем, явно осуществляйте проверку этого случая и обрабатывайте его соответствующим образом (например, выводите сообщение об ошибке), и не допускайте возникновения фатальной ошибки.

3.8. В цикле, управляемом контрольным значением, подсказки для запроса ввода данных должны явно напоминать пользователю, чему равно это значение.

3.9. Не проверяйте на равенство значения с плавающей точкой.

3.10. Унарные операции следует помещать непосредственно рядом с их операндами без разделительных пробелов.

3.11. Стандартом ANSI в общем случае не специфицируется порядок, в котором будут оцениваться операнды той или иной операции (хотя в главе 4 мы увидим, что для некоторых операций из этого правила существуют исключения). Поэтому программист должен избегать команд с операциями инкремента или декремента, в которых некоторая переменная подвергается их воздействию более одного раза.

Советы по повышению эффективности

3.1. Инициализация переменных при их объявлении уменьшает время выполнения программы.

3.2. Выражение с операцией присваивания (например, **c += 3**) компилируется быстрее, чем эквивалентное раскрытое выражение (**c = c + 3**), поскольку переменная c в первом выражении оценивается только один раз, в то время как во втором выражении она оценивается дважды.

3.3. Многие из советов по повышению эффективности, которые мы даем в этой книге, приводят к незначительным на первый взгляд улучшениям, поэтому у читателя может возникнуть искушение игнориро-

вать их. Дело в том, что заставить программу выполняться значительно быстрее может только совокупный эффект от всех этих мер по повышению эффективности. Кроме того, значительное улучшение достигается при внесении вроде бы незначительного усовершенствования в цикл, который может повторяться большое количество раз.

Общие методические замечания

3.1. Составной оператор может быть помещен в любое место программы, где может стоять простой оператор.

3.2. Подобно тому, как составной оператор можно помещать всюду, где может быть помещен простой оператор, так же возможно не помещать там никакого оператора вообще, то есть использовать пустой оператор. Пустой оператор представляется помещением точки с запятой (;) на место, обычно занимаемое некоторым оператором.

3.3. Каждое уточнение, так же как и сам верхний уровень, представляет собой полную спецификацию алгоритма; меняется только уровень детализации.

3.4. Многие программы логически могут быть разделены на три этапа: этап инициализации, на котором происходит инициализация переменных программы; этап обработки, на котором происходит ввод данных и соответствующая настройка переменных программы; и этап завершения работы программы, на котором происходит вычисление и вывод окончательных результатов.

3.5. Программист завершает процесс нисходящего пошагового уточнения в тот момент, когда алгоритм на псевдокоде специфицирован достаточно подробно для того, чтобы программист мог преобразовать псевдокод в программу на С. Реализация программы на С в этом случае обычно не вызывает затруднений.

3.6. Опыт показывает, что наиболее трудной частью решения задачи на компьютере является разработка алгоритма ее решения. Как только соответствующий алгоритм построен, процесс написания работающей программы на С обычно не вызывает затруднений.

3.7. Многие программисты пишут программы, никогда не пользуясь никакими инструментами разработки программ типа псевдокода. Они полагают, что поскольку их конечной целью является решение задачи на компьютере, написание псевдокода только задерживает производство конечного продукта.

Упражнения для самоконтроля

3.1. Ответьте на каждый из следующих вопросов.

a) Процедура решения задачи в виде действий, которые надлежит выполнить, и порядка, в котором эти действия должны быть выполнены, называется _____.

b) Задание порядка, в котором компьютер выполняет операторы, называется _____.

c) Все программы могут быть написаны с использованием трех управляющих структур: _____, _____ и _____.

d) Структура выбора _____ используется для выполнения некоторого действия в случае истинности условия и для выполнения другого действия в случае его ложности.

e) Несколько операторов, сгруппированных вместе в фигурных скобках ({ и }), называются _____.

f) Структура повторения _____ специфицирует многократное выполнение оператора или группы операторов, пока некоторое условие остается истинным.

g) Повторение набора команд определенное число раз называется повторением, _____.

h) Если заранее неизвестно, сколько раз будет повторено некоторый набор операторов, то для завершения повторения может быть использовано _____ значение.

3.2. Напишите четыре различных оператора C, каждый из которых прибавляет 1 к целочисленной переменной **x**.

3.3. Напишите одиночный оператор C для выполнения каждой из следующих задач:

a) Присвойте сумму **x** и **y** переменной **z** и увеличьте значение переменной **x** на 1 после проведения вычисления.

b) Умножьте значение переменной **product** на 2, используя операцию ***=**.

c) Умножьте значение переменной **product** на 2, используя операции **=** и ***.**

d) Проверьте, является ли значение переменной **count** больше 10. Если является, выведите на экран **«Значение переменной count больше 10»**.

e) Уменьшите значение переменной **x** на 1, затем вычтите его из переменной **total**.

f) Прибавьте значение переменной **x** к переменной **total**, затем уменьшите **x** на 1.

g) Вычислите остаток от деления переменной **q** на значение переменной **divisor** и присвойте результат переменной **q**. Напишите этот оператор двумя различными способами.

h) Выведите значение **123.4567** с точностью до **2** знаков. Какое значение будет выведено?

i) Выведите значение с плавающей точкой, равное **3.14159**, с тремя знаками после десятичной точки. Какое значение будет выведено?

3.4. Напишите оператор C для выполнения каждой из следующих задач.

a) Объявите переменные **sum** и **x** с типом **int**.

b) Инициализируйте переменную **x** единицей.

c) Инициализируйте переменную **sum** нулем.

d) Прибавьте переменную **x** к переменной **sum** и присвойте результат переменной **sum**.

e) Выведите на экран строку **«Сумма равна: »**, за которой следует значением переменной **sum**.

3.5. Объедините операторы, которые вы написали в упражнении 3.4, в программу, которая вычисляет сумму целых чисел от 1 до 10. Используйте структуру **while** для выполнения в цикле указанного вычисления и инкрементных операций. Цикл должен завершиться, когда значение переменной **x** станет равным 11.

3.6. *Определите значение каждой из переменных после выполнения вычисления. Предположите, что перед началом выполнения каждого оператора все переменные равны 5.*

a) `product *= x++;`

b) `result = ++x + x;`

3.7. Напишите одиночный оператор C, который

a) Вводит целочисленную переменную **x** с помощью **scanf**.

b) Вводит целочисленную переменную **y** с помощью **scanf**.

c) Инициализирует целочисленную переменную **i** единицей.

d) Инициализирует целочисленную переменную **power** единицей.

e) Умножает переменную **power** на **x** и присваивает результат переменной **power**.

f) Увеличивает переменную **y** на **1**.

g) Проверяет переменную **y**, чтобы выяснить, является ли она меньшей или равной **x**.

h) Выводит целочисленную переменную **power** с помощью **printf**.

3.8. Напишите программу на C, которая использует операторы упражнения 3.7 для расчета значения **x** в степени **y**. Программа должна включать управляющую структуру повторения **while**.

3.9. Найдите и исправьте ошибки в каждом из следующих случаев:

a)
```
while (c <= 5) {
   product *= c;
   ++c;
```

b)
```
scanf("%.4f", &value);
```

c)
```
if (gender == 1)
   printf("Woman\n");
else;
   printf("Man\n");
```

3.10. Что неправильно в следующей структуре повторения **while**?

```
while (z >= 0)
   sum += z;
```

Ответы на упражнения для самоконтроля

3.1. a) *Алгоритмом*. b) *Программным управлением*. c) *Последовательность, выбор, повторение*. d) **if/else**. e) Составным оператором. f) **while**. g) Управляемым счетчиком. h) Контрольное.

3.2.
```
x = x + 1;
x += 1;
++x;
x++;
```

3.3. a)
```
z = x++ + y;
```

b)
```
product *= 2;
```

c)
```
product = product * 2;
```

d)
```
if (count > 10)
   printf ("Count is greater than 10.\n");
```

e)
```
total -= --x;
```

f)
```
total += x--;
```

g)
```
q %= divisor;
q = q %= divisor;
```

h)
```
printf("%.2f", 123.4567);
```

На экран выводится **123.46**.

i)
```
printf ("%.3f\n", 3.14159);
```

На экран выводится **3.142**.

3.4. a)
```
int sum, x;
```

b)
```
x = 1;
```

c)
```
sum = 0;
```

d)
```
sum += x; or sum = sum + x;
```

e)
```
printf("The sum is: %d\n", sum);
```

3.5.
```
/* Вычисляет сумму целых чисел от 1 до 10 */
#include <stdio.h>

main()
{
   int sum, x;

   x = 1;
   sum = 0;
   while (x <= 10) {
      sum += x;
      ++x;
   }

   printf ("The sum is: %d\n", sum);
}
```

3.6. a)
```
product = 25, x = 6;
```

b)
```
result = 12, x = 6;
```

3.7. a)
```
scanf("%d", &x);
```

b)
```
scanf("%d", &y);
```

c)
```
i = 1;
```

d)
```
power = 1;
```

e) `power *= x;`

f) `y++;`

g) `if(y <= x)`

h) `printf ("%d", power);`

3.8.
```
/* возводит x в степень y */
#include <stdio.h>

main()
   int x,y,i,power;

   i = 1;
   power = 1;
   scanf("%d", &x);
   scanf(%"d", &y);

while (i <= y) {
      power *= x;
      ++i;
   }

   printf ("%d",power);
   return 0;
}
```

3.9. a) Ошибка: отсутствие завершающей правой фигурной скобки в теле структуры **while**.

Исправление: добавьте завершающую правую фигурную скобку после оператора **++c;**.

b) Ошибка: точность, используемая в спецификации преобразования функции **scanf**.

Исправление: удалите **.4** из спецификации преобразования.

c) Ошибка: помещение точки с запятой после ветви **else** структуры **if/else** приводит к логической ошибке. Второй оператор **printf** будет выполняться всегда.

Исправление: удалите точку с запятой после **else**.

3.10. Значение переменной **z** ни разу не изменяется в структуре **while**. Следовательно, возникает бесконечный цикл. Для недопущения бесконечного цикла переменная **z** должна уменьшаться, так чтобы в конечном итоге ее значение стало равно **0**.

Упражнения

3.11. Найдите и исправьте ошибки в каждом из следующих случаев (Замечание: в каждом фрагменте кода может быть более одной ошибки):

a)
```
if (age >= 65);
   printf ("Age is greater than or equal to 65\n");
else
   printf ("Age is less than 65\n");
```

b)
```
int x = 1, total;
while (x <=10) {
   total += x;
   ++x;
}
```

c)
```
While (x <= 100)
   total =+ x;
   ++x;
```

d)
```
while (y > 0) {
   printf("%d\n", y)
   ++y;
}
```

3.12. Заполните пропуски в каждом из следующих утверждений:

a) Решение любой задачи включает выполнение ряда действий в определенном _____.

b) Синонимом для процедуры является _____.

c) Переменная, в которой накапливается сумма нескольких чисел, называется _____.

d) Процедура присвоения некоторым переменным определенных значений в начале программы называется _____.

e) Специальное значение, используемое для указания на «конец ввода данных», называется _____, _____, _____ или _____ значением.

f) _____ является графическим представлением алгоритма.

g) В блок-схеме порядок, в котором должны выполняться шаги алгоритма, обозначается символами _____.

h) Конечный символ указывает на _____ и _____ любого алгоритма.

i) Символы прямоугольников соответствуют вычислениям, которые обычно выполняются с помощью операторов _____, и операциям ввода/вывода, которые обычно выполняются путем обращений к стандартным библиотечным функциям _____ и _____.

j) Элемент, который помещается внутри символа принятия решения, называется _____.

3.13. Что выводит следующая программа?

```
main()
{
   int x = 1, total = 0, y;
   while (x <= 10) {
      y = x * x;
      printf( "%d\n", y);
      total += y;
      ++x;
   }
   printf ("Total is %d\n", total);
   return 0;
}
```

3.14. Напишите одиночные операторы псевдокода, которые соответствуют каждому из следующих действий:

a) Отобразите на экране сообщение **«Введите два числа»**.

b) Присвойте сумму переменных **x**, **y** и **z** переменной **p**.

c) Следующее условие должно быть проверено в структуре выбора **if/else**: текущее значение переменной m больше удвоенного текущего значения переменной **v**.

d) Введите значения для переменных **s**, **r** и **t** с клавиатуры.

3.15. Сформулируйте алгоритм на псевдокоде для каждого из следующих действий:

a) Введите с клавиатуры два числа, вычислите их сумму и отобразите результат вычислений.

b) Введите с клавиатуры два числа, определите и отобразите на экране, какое (если какое-то) из этих двух чисел больше.

c) Введите с клавиатуры последовательность положительных чисел, определите и отобразите на экране сумму этих чисел. Предположите, что пользователь вводит контрольное значение **-1** для указания на «конец ввода данных».

3.16. Установите, какие из следующих высказываний являются истинными и какие — ложными. Если утверждение является ложным, объясните почему.

a) Опыт показывает, что наиболее трудной частью решения задачи на компьютере является создание работающей программы на C.

b) В качестве контрольного должно использоваться значение, которое нельзя спутать с допустимым значением данных.

c) Линии перехода указывают на действия, подлежащие выполнению.

d) Условия, записываемые внутри символов принятия решения, всегда содержат арифметические операции (т.е. +, -, *, / и %).

e) При нисходящем пошаговом уточнении каждое уточнение является полным представлением алгоритма.

Для упражнений с 3.17 по 3.21 выполните каждый из следующих шагов:

1. Прочитайте постановку задачи.
2. Сформулируйте алгоритм, используя для этого псевдокод и нисходящее пошаговое уточнение.
3. Напишите программу на C.
4. Протестируйте, отладьте и выполните программу на C.

3.17. Из-за высокой цены на бензин водители интересуются пробегом своих автомобилей. Некий водитель отследил несколько заправок своего автомобиля, записывая пройденное расстояние в милях и объем каждой заправки в галлонах. Разработайте программу на C для ввода пройденного расстояния в милях и объема каждой заправки в галлонах. Программа должна вычислять и отображать на экране количество пройденных миль на галлон для каждой заправки автомобиля. После обработки всей входной информациипрограмма должна

вычислить и вывести общее количество пройденных миль на галлон по всем заправкам.

```
Введите расход бензина (-1, если ввод закончен): 12.8
Введите пройденный путь: 287
Для этой заправки получено миль/галлон 22.421875

Введите расход бензина (-1, если ввод закончен): 10.3
Введите пройденный путь: 200
Для этой заправки получено миль/галлон 19.417475

Введите расход бензина (-1, если ввод закончен): 5
введите пройденный путь: 120
Для этой заправки получено миль/галлон 24.000000

Введите расход бензина (-1, если ввод закончен): -1

Среднее число миль/галлон 21.601423
```

3.18. Разработайте программу на С, которая будет определять, превысил ли тот или иной покупатель универмага предельный размер кредита на своем расчетном счете. Для каждого покупателя доступна следующая информация:

1. Номер счета
2. Баланс на начало месяца
3. Общее сумма по всем статьям расхода для данного покупателя в этом месяце
4. Общее сумма всех кредитов, отнесенный на счет данного покупателя в этом месяце
5. Допустимый предельный размер кредита

Программа должна вводить все эти данные, вычислять новый баланс (= начальный баланс + расходы - кредит) и определять, превышает ли новый баланс предельный размер кредита покупателя. Для тех покупателей, чей предельный размер кредита превышен, программа должна отображать на экране номер счета покупателя, предельный размер кредита, новый баланс и сообщение «Предельный размер кредита превышен».

```
Введите номер счета (-1, если ввод закончен): 100
Введите начальный баланс: 5394.78
Введите общую сумму расходов: 1000.00
Введите общую сумму кредита: 500.00
Введите предельный размер кредита: 5500.00
Счет: 100
Предельный размер кредита: 5500.00
Баланс: 5894.78
Предельный размер кредита превышен.

Введите номер счета (-1, если ввод закончен): 200
Введите начальный баланс: 1000.00
Введите общую сумму расходов: 123.45
Введите общую сумму кредита: 321.00
Введите предельный размер кредита: 1500.00
```

```
Введите номер счета (-1, если ввод закончен): 300
Введите начальный баланс: 500.00
Введите общую сумму расходов: 274.73
Введите общую сумму кредита: 100.00
Введите предельный размер кредита: 80

Введите номер счета (-1, если ввод закончен): -1
```

3.19. Некая крупная химическая компания платит своим продавцам на основе комиссионных вознаграждений. Продавцы получают $200 в неделю плюс 9 процентов от их валовых продаж за эту неделю. Например, продавец, реализующий за неделю химических препаратов на $5000, получает $200 плюс 9 процентов от $5000, или в сумме $650. Разработайте программу на C для ввода валовых продаж для каждого продавца за последнюю неделю и расчета и отображения на экране заработка этого продавца. Обрабатывайте за один раз данные для одного продавца.

```
Введите сумму продаж в долларах (-1, если ввод закончен):
 5000.00
Зарплата: $650.00

Введите сумму продаж в долларах (-1, если ввод закончен):
 1234.56
Зарплата: $311.11

Введите сумму продаж в долларах (-1, если ввод закончен):
 1088.89
Зарплата: $298.00

Введите сумму продаж в долларах (-1, если ввод закончен): -1
```

3.20. Простые проценты по ссуде рассчитываются по формуле

```
interest = principal * rate * days / 365
```

В предыдущей формуле принимается, что **rate** является процентной ставкой за год и поэтому включает деление на 365 (дней). Разработайте программу на C, которая будет вводить значения **principal**, **rate** и **days** для нескольких ссуд и будет вычислять и отображать на экране простые проценты по каждой ссуде, используя для этого предыдущую формулу.

```
Введите основную сумму ссуды (-1, если ввод закончен): 1000.00
Введите процентную ставку: .1
Введите срок ссуды в днях: 365
Выплаты по простым процентам составляют $100.00

Введите основную сумму ссуды (-1, если ввод закончен): 1000.00
Введите процентную ставку: .08375
Введите срок ссуды в днях: 224
Выплаты по простым процентам составляют $51.40

Введите основную сумму ссуды (-1, если ввод закончен): 10000.00
Введите процентную ставку: .09
Введите срок ссуды в днях: 1460
Выплаты по простым процентам составляют $3600.00

Введите основную сумму ссуды (-1, если ввод закончен): -1
```

3.21. Разработайте программу на C для определения общей зарплаты для каждого из нескольких служащих. Компания платит «обычную зарплату» за первые 40 часов, отработанных каждым служащим, и «полуторную зарплату» за все время, отработанное сверх 40 часов. Вам предоставлены список служащих компании, число часов, отработанных каждым служащим за последнюю неделю, и почасовой тариф каждого служащего. Ваша программа должна ввести для каждого служащего эту информацию и определить и отобразить на экране его общую зарплату.

```
Введите # отработанных часов (-1, если ввод закончен): 39
Введите почасовой тариф работника ($00.00): 10.00
Зарплата составляет $390.00

Введите # отработанных часов (-1, если ввод закончен): 40
Введите почасовой тариф работника ($00.00): 10.00
Зарплата составляет $400.00

Введите # отработанных часов (-1, если ввод закончен): 41
Введите почасовой тариф работника ($00.00): 10.00
Зарплата составляет $415.00

Введите # отработанных часов (-1, если ввод закончен): -1
```

3.22. Напишите программу на C, показывающую различие между операциями предекремента и постдекремента, используя для этого операцию декремента --.

3.23. Напишите программу на C, которая выводит в цикле числа от 1 до 10, располагая их бок о бок в одной и той же строке и разделяя тремя пробелами.

3.24. Процесс нахождения наибольшего числа (т.е. максимума группы чисел) часто используется в компьютерных приложениях. Например, программа для определения победителя конкурса продаж вводит количество единиц товара, проданных каждым продавцом. Продавец, реализовавший наибольшее количество единиц товара, побеждает в конкурсе. Напишите программу на псевдокоде, а затем программу на C для ввода серии из 10 чисел и определения и вывода на печать наибольшего из этих чисел. Подсказка: ваша программа должна использовать такие переменные:

counter; счетчик до 10 (т.е. для отслеживания количества введенных чисел и определения момента, когда обработаны все 10 чисел).

number; текущее число, введенное в программу.

largest; наибольшее число, найденное до сих пор.

3.25. Напишите программу на C, которая выводит в цикле следующую таблицу значений:

```
N       10*N    100*N   1000*N

1       10      100     1000
2       20      200     2000
3       30      300     3000
4       40      400     4000
```

```
5       50      500     5000
6       60      600     6000
7       70      700     7000
8       80      800     8000
9       90      900     9000
10      100     1000    10000
```

Для отделения столбцов табуляциями в операторе **printf** может быть использован символ табуляции **\t**.

3.26. Напишите программу на C, которая порождает в цикле следующую таблицу значений:

```
A       A+2     A+4     A+6

3       5       7       9
6       8       10      12
9       11      13      15
12      14      16      18
15      17      19      21
```

3.27. Применяя подход, аналогичный применявшемуся в упражнении 3.24, найдите из 10 чисел два наибольших значения. Замечание: вы можете вводить каждое число только один раз.

3.28. Измените программу на рис. 3.10 так, чтобы она проверяла правильность своих входных данных. После ввода любого значения, отличного от 1 или 2, продолжайте выполнение цикла до тех пор, пока пользователь не введет правильного значения.

3.29. Что выводит следующая программа?

```
#include <stdio.h>

main()
{
   int count = 1;
   while (count <= 10) {
      printf("%s\n", count % 2 ? "****" :"++++++++");
      ++count;
   }

   return 0;
}
```

3.30. Что выводит следующая программа?

```
#include <stdio.h>

main()
{
   int row = 10, column;

   while (row >= 1) {
      column = 1;

      while (column <= 10) {
         printf ("%s", row % 2 ? "<" : ">");
         ++column;
```

```
        }

        --row;
        printf"\n");
    }

    return 0;
}
```

3.31. *(Проблема висящего else)* Определите результаты вычислений для каждого из следующих фрагментов кода, когда **x** равен **9** и **y** равен **11**, и когда **x** равен **11** и **y** равен **9**. Обратите внимание, что компилятор игнорирует отступы в программе на C. Кроме того, компилятор C всегда ассоциирует **else** с последним предшествующим ему **if**, если не указано иначе посредством фигурных скобок **{}**. Поскольку на первый взгляд программист может и не быть уверенным, какому **if** соответствует данный **else**, это получило название проблемы «висящего else». Мы удалили отступы из следующего кода, чтобы еще больше усложнить задачу. (Подсказка: примените соглашения об отступах, которые вы изучили.)

a)
```
if (x < 10)
if (y > 10)
printf ("*****\n");
else
printf ("#####\n");
printf ("$$$$$\n");
```

b)
```
if (x < 10) {
if (y > 10)
printf ("*****\n");
}
else {
printf ("#####\n");
printf ("$$$$$\n");
}
```

3.32. *(Еще одна проблема висящего else)* Измените следующий код для получения показанного вывода. Используйте соответствующие методы выравнивания. Возможно, вам не придется делать никаких изменений, кроме вставки фигурных скобок. Компилятор игнорирует выравнивание в программах на C. Мы удалили из кода отступы, чтобы еще больше усложнить задачу. Замечание: возможно, не потребуется никаких изменений.

```
if (y == 8)
if (x == 5)
printf ("@@@@@\n")
else
printf ("#####\n");
printf ("$$$$$\n");
printf ("&&&&&\n");
```

a) В предположении, что **x = 5** и **y = 8**, программа должна вывести:

```
@@@@@
$$$$$
&&&&&
```

b) В предположении, что **x = 5** и **y = 8**, программа должна вывести:

```
@@@@@
```

c) В предположении, что **x = 5** и **y = 8**, программа должна вывести:

```
@@@@@
&&&&&
```

d) В предположении, что **x = 5** и **y = 7**, должен быть получен следующий вывод. Замечание: все три последних оператора **printf** являются частью составного оператора.

```
#####
$$$$$
&&&&&
```

3.33. Напишите программу, которая считывает размер стороны квадрата и затем выводит этот квадрат в виде звездочек. Ваша программа должна работать для всех квадратов с размерами сторон между 1 и 20. Например, если ваша программа считывает размер, равный 4, она должна вывести

```
****
****
****
****
```

3.34. Измените программу, которую вы написали в упражнении 3.33 так, чтобы она выводила полый квадрат. Например, если ваша программа считывает размер, равный 5, она должна вывести

```
*****
*   *
*   *
*   *
*****
```

3.35. Палиндромом называется число или фраза текста, которая читается одинаково как слева направо, так и в обратном порядке. Например, каждое из следующих пятизначных целых чисел является палиндромом: 12321, 55555, 45554 и 11611. Напишите программу, которая считывает пятизначное целое число и определяет, является ли оно палиндромом. (Подсказка: примените операции деления и взятия по модулю для разложения числа на отдельные цифры.)

3.36. Введите целое число, содержащее только 0 и 1 (т.е. «двоичное» целое число) и выведите его десятичный эквивалент. (Подсказка: примените операции деления и взятия по модулю для отделения справа налево одного за другим разрядов «двоичного» числа. Подобно тому, как в десятичной системе счисления цифра самого правого разряда имеет позиционное значение 1, следующая из оставшихся цифр имеет позиционное значение 10, потом 100, потом 1000 и т.д., в системе двоичного счисления цифра самого правого разряда имеет позиционное значение 1, следующая из оставшихся цифр имеет позиционное значение 2, потом 4, потом 8 и т.д. Таким образом, десятичное число 234 может быть интерпретировано как 4 * 1 + 3 * 10 + 2 * 100. Десятичным эквивалентом двоичного 1101 является 1 * 1 + 0 * 2 + 1 * 4 + 1 * 8, или 1 + 0 + 4 + 8, или 13.)

3.37. Мы все время слышим о том, как быстры компьютеры. Каким образом вы можете определить, насколько быстро в действительности функционирует ваша собственная машина? Напишите программу с циклом **while**, который считает от 1 до 3 000 000 с шагом 1. Всякий раз, когда результат счета становится кратным 1 000 000, выводите это значение на экран. По своим часам рассчитайте время, требуемое для выполнения каждого миллиона повторений цикла.

3.38. Напишите программу, которая выводит 100 звездочек, по одной за один раз. После каждой десятой звездочки ваша программа должна выводить символ новой строки. (Подсказка: считайте от 1 до 100. Примените операцию взятия по модулю для распознавания всех случаев, когда счетчик становится кратным 10.

3.39. Напишите программу, которая считывает целое число и определяет (выводя результат на печать), сколько цифр в этом числе равно 7.

3.40. Напишите программу, которая отображает на экране следующий рисунок шахматной доски:

```
* * * * * * * *
 * * * * * * * *
* * * * * * * *
 * * * * * * * *
* * * * * * * *
 * * * * * * * *
* * * * * * * *
 * * * * * * * *
```

Ваша программа может использовать только три оператора **printf**, один в виде

```
printf( "* ");
```

другой

```
printf(" ");
```

и третий

```
printf ("\n");
```

3.41. Напишите программу, которая последовательно выводит числа, кратные целому числу 2, а именно 2, 4, 8, 16, 32, 64 и т.д. Ваш цикл не должен завершаться (т. е. вы должны создать бесконечный цикл). Что произойдет, когда вы будете выполнять эту программу?

3.42. Напишите программу, которая считывает радиус круга (в виде значения типа **float**), вычисляет и выводит значения его диаметра, периметра и площади. Примите для π значение 3.14159.

3.43. Что неправильно в следующем операторе? Перепишите оператор, чтобы он выполнял то, что, возможно, хотел сделать программист.

```
printf ("%d", ++(x + y));
```

3.44. Напишите программу, которая считывает три ненулевых значения типа **float** и определяет, могут ли они представлять стороны треугольника, выводя результат на экран.

3.45. Напишите программу, которая считывает три ненулевых целых числа и определяет, могут ли они быть сторонами прямоугольного треугольника, выводя результат на экран.

3.46. Компании необходимо передавать данные по телефону, но есть опасение, что телефоны могут прослушиваться. Все данные передаются в виде четырехзначных целых чисел. Представители компании попросили вас написать программу для шифровки их данных, чтобы сделать их передачу более надежной. Ваша программа должна считывать четырехзначное целое число и шифровать его следующим образом: «Заменить каждую цифру значением *(суммы этой цифры и 7) по модулю 10.* Затем поменять местами первую цифру и третью, вторую цифру и четвертую». Затем вывести зашифрованное целое число. Напишите отдельную программу, которая вводит зашифрованное четырехзначное целое число и дешифрует его, воссоздавая первоначальное число.

3.47. Факториал неотрицательного целого числа n записывается в виде $n!$ (произносится «n факториал») и определяется следующим образом:

$n! = n \cdot (n-1) \cdot (n-2) \cdot$К (для значений n, больших или равных 1)

и

$n! = 1$ (для n=0).

Например, $5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1$, что равняется 120.

a) Напишите программу, которая считывает неотрицательное целое число, вычисляет и выводит его факториал.

b) Напишите программу, которая оценивает значение математической константы e по формуле:

$$e = 1 + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \text{К}$$

c) *Напишите программу, которая вычисляет значение* e^x по формуле

$$e^x = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \text{К}$$

Управление программой



Цели

- Научиться применению структур повторения **for** и **do/while**.
- Изучить структуру множественного выбора **switch**.
- Научиться применению операторов управления **break** и **continue**.
- Освоить использование логических операций.

Содержание

4.1. Введение
4.2. Основы структур повторения
4.3. Повторение, управляемое счетчиком
4.4. Структура повторения for
4.5. Структура for: замечания и рекомендации
4.6. Примеры использования структуры for
4.7. Структура со множественным выбором switch
4.8. Структура повторения do/while
4.9. Операторы break и continue
4.10. Логические операции
4.11. Смешивание операций равенства (==) и присваивания (=)
4.12. Краткая сводка по структурному программированию

Резюме • Распространенные ошибки программирования • Хороший стиль программирования • Советы по повышению эффективности • Советы по переносимости программ • Общие методические замечания • Упражнения для самоконтроля • Ответы к упражнениям для самоконтроля • Упражнения

4.1. Введение

К настоящему моменту читатель должен чувствовать себя уверенно в написании простых, но вместе с тем законченных программ на языке С. В этой главе будет более подробно рассмотрено повторение, в частности, будут представлены дополнительные управляющие структуры повторения, а именно структура **for** и структура **do/while**. Будет введена структура со множественным выбором **switch**. Мы обсудим оператор **break** для безотлагательного и быстрого выхода из некоторых управляющих структур и оператор **continue** для пропуска оставшихся операторов тела структуры повторения и перехода к следующей итерации цикла. В этой главе обсуждаются логические операции, используемые для объединения условий, а заканчивается она кратким обзором принципов структурного программирования, представленных в главах 3 и 4.

4.2. Основы структур повторения

Большинство программ содержит повторение, т.е. *циклы*. Цикл — это группа команд, многократно выполняемых компьютером, пока некоторое *условие продолжения* цикла остается истинным. До сих пор мы обсудили два способа повторений:

1. Повторение, управляемое счетчиком

2. Повторение, управляемое контрольным значением

Повторение, управляемое счетчиком, иногда называют *определенным повторением*, поскольку мы в точности знаем заранее, сколько раз будет выполнен цикл. Повторение, управляемое контрольным значением, иногда называют *неопределенным повторением*, поскольку заранее неизвестно, сколько раз потребуется выполнить цикл.

В повторении, управляемом счетчиком, для подсчета числа повторений используется управляющая переменная. Управляющая переменная увеличивается (обычно на 1) всякий раз, когда выполняется тело цикла. Когда значение управляющей переменной показывает, что было выполнено соответствующее число повторений, цикл завершается, и компьютер продолжает выполнение программы с оператора, следующего за структурой повторения.

Контрольные значения используются для управления повторением в тех случаях, когда:

1. Заранее неизвестно точное число повторений, и

2. Цикл содержит операторы, которые получают данные при каждом выполнении цикла.

Контрольное значение служит признаком «конца данных». Оно вводится после того, как в программу были переданы все обычные элементы данных. Контрольные значения должны отличаться от возможных значений данных.

4.3. Повторение, управляемое счетчиком

Для повторения, управляемого счетчиком, требуется:

1. *Имя* управляющей переменной (или счетчика цикла).

2. *Начальное значение* управляющей переменной.

3. Значение *инкремента* (приращения со знаком плюс) или декремента (приращения со знаком минус), на которое управляющая переменная изменяет свое значение после каждого выполнения цикла.

4. Условие, в котором происходит проверка на *конечное значение* управляющей переменной (т.е. должно ли продолжаться выполнение цикла).

Рассмотрим простую программу, показанную на рис. 4.1, которая выводит числа от 1 до 10. Объявление

```
int counter = 1;
```

именует управляющую переменную **(counter)**, объявляет ее целым числом, резервирует для нее память и присваивает ей начальное значение, равное **1**. Это объявление не является исполняемым оператором.

```
/* Повторение, управляемое счетчиком */
#include <stdio.h>

main()
{
   int counter = 1;                    /* инициализация */

   while (counter <= 10) {             /* условие повторения */
      printf("%d\n", counter);
      ++counter;                       /* приращение */
   }

   return 0;
}
```

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

Рис. 4.1. Повторение, управляемое счетчиком

Объявление и инициализация счетчика могли бы быть выполнены с помощью операторов

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

Объявление не является исполняемым оператором, но присваивание является таковым. Мы используем оба метода инициализации переменных.

Оператор

```
++counter;
```

увеличивает счетчик цикла на 1 при каждом выполнении цикла. Условие продолжения цикла структуры **while** проверяет, является ли значение управляющей переменной меньшим или равным **10** (это последнее значение, для которого условие истинно). Обратите внимание, что тело этого цикла **while** выполняется и в том случае, когда управляющая переменная равна **10**. Цикл завершается, когда управляющая переменная становится больше **10** (т.е. переменная **counter** становится равной **11**).

Программирующие на C сделали бы программу на рис. 4.1 более краткой, инициализировав переменную **counter** значением **0** и заменив структуру **while** на

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

Этот код экономит один оператор, поскольку приращение выполняется непосредственно в условии структуры **while** перед его проверкой. Кроме того, здесь устранены фигурные скобки вокруг тела структуры **while**, поскольку **while** теперь содержит только один оператор. Написание кода в такой сжатой манере требует некоторой практики.

Распространенная ошибка программирования 4.1

Ввиду того, что значения с плавающей точкой могут быть неточными, управление циклами со счетчиком при помощи переменных с плавающей точкой может привести к неточным значениям счетчика и ошибочным результатам проверки на окончание цикла.

Хороший стиль программирования 4.1

Управляйте циклами со счетчиком с помощью целочисленных значений.

Хороший стиль программирования 4.2

Делайте отступы для операторов в теле каждой управляющей структуры.

Хороший стиль программирования 4.3

Помещайте пустую строку до и после каждой управляющей структуры большого размера для выделения ее в тексте программы.

Хороший стиль программирования 4.4

Слишком большое число уровней вложенности может сделать программу трудной для понимания. Пытайтесь избегать использования более трех уровней вложенности.

Хороший стиль программирования 4.5

Сочетание междустрочного интервала до и после управляющих структур и отступов в теле управляющих структур в пределах их заголовков придает программам двумерный вид, который значительно повышает их удобочитаемость.

4.4. Структура повторения for

Структура повторения **for** автоматически контролирует все детали повторения, управляемого счетчиком. Для иллюстрации всей мощи цикла **for** давайте перепишем программу на рис. 4.1. Результат показан на рис. 4.2.

Программа работает следующим образом. Когда структура **for** начинает выполняться, управляющая переменная **counter** инициализируется значением **1**. Затем проверяется условие продолжения цикла **counter <= 10**. Поскольку начальное значение переменной **counter** равно **1**, условие удовлетворяется и оператор **printf** выводит значение переменной **counter**, а именно **1**.

Затем в выражении **counter++** происходит приращение управляющей переменной counter и снова начинается выполнение цикла с проверки условия его продолжения. Поскольку управляющая переменная теперь равна **2**, конечное значение не превышено, и поэтому программа снова выполняет оператор **printf**. Этот процесс продолжается до тех пор, пока управляющая переменная **counter** не увеличивается до ее конечного значения 11 — это приводит к неудаче при проверке условия продолжения цикла, и повторение завершается. Выполнение программы продолжается с первого оператора после структуры **for** (в нашем случае с оператора **return** в конце программы).

```
/* Повторение, управляемое счетчиком с использованием
   структуры for */
#include <stdio.h>

main()
{
   int counter = 1;

   /* инициализация, условие повторения и приращение */
   /* все они включены в заголовок структуры for */
   for (counter = 1; counter <= 10; counter++)
      printf("%d\n", counter);

   return 0;
}
```

Рис. 4.2. Повторение, управляемое счетчиком с использованием структуры **for**

На рис. 4.3 более подробно рассмотрена структура **for**, представленная на рис. 4.2. Обратите внимание, что структура **for** «делает всю работу» — в ней специфицированы все элементы, необходимые для повторения, управляемого счетчиком с использованием управляющей переменной. Если в теле цикла **for** содержится более одного оператора, для определения тела цикла требуются фигурные скобки.

Обратите внимание, что на рис. 4.2 используется условие продолжение цикла **counter <= 10**. Если бы программист по ошибке написал **counter < 10**, то цикл был бы выполнен только 9 раз. Это является распространенной логической ошибкой, называемой ошибкой *смещения счетчика*.

Распространенная ошибка программирования 4.2

Использование неправильной операции отношения или неправильного конечного значения счетчика цикла в условии структур **while** или **for** может привести к ошибке смещения счетчика.

Хороший стиль программирования 4.6

Использование конечного значения в условии структур **while** или **for** и операции отношения **<=** помогает избежать ошибок сдвига счетчика. Например, для цикла, используемого для вывода значений от 1 до 10, условием продолжения цикла должно быть **counter <= 10**, а не **counter < 11** или **counter < 10**.

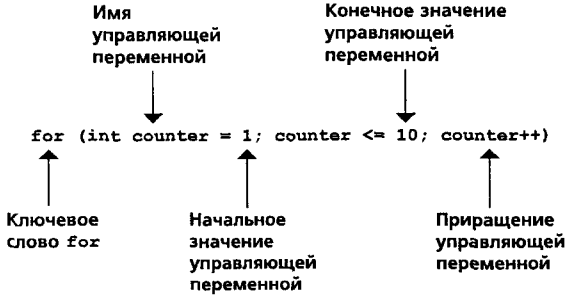


Рис. 4.3. Компоненты типичного заголовка структуры **for**

Общий формат структуры **for**

```
for (выражение1; выражение2; выражение3)
    оператор
```

где *выражение1* инициализирует переменную управления циклом, *выражение2* является условием продолжения цикла и *выражение3* служит для приращения управляющей переменной. В большинстве случаев структура **for** может быть представлена эквивалентной структурой **while** следующим образом:

```
выражение1;
while (выражение2) {
   оператор
   выражение3;
}
```

Из этого правила есть исключение, которое мы обсудим в разделе 4.9.

Часто *выражение1* и *выражение3* являются списками выражений, разделенных запятыми. Запятые здесь на самом деле являются *операциями-запятыми*, которые гарантируют, что списки выражений будут оцениваться слева направо. Значением и типом списка выражений, разделенных запятыми, являются значение и тип самого правого выражения в списке. Операция-запятая чаще всего используется в структуре **for**. Ее основным назначением является реализация множественной инициализации и/или нескольких выражений приращения. Например, в одной структуре **for** могут быть две управляющих переменных, которые должны инициализироваться и получать приращение.

Хороший стиль программирования 4.7

Помещайте в разделы инициализации и приращения структуры **for** только выражения, включающие управляющие переменные. Манипуляции с другими переменными должны производиться либо перед циклом (если они выполняются только один раз, как операторы инициализации), либо в теле цикла (если они выполняются по одному разу за повторение, как операторы инкремента или декремента).

Все три выражения в структуре **for** являются необязательными. Если опущено *выражение2*, С предполагает, что условие истинно, и возникает бесконечный цикл. *Выражение1* может быть опущено, если управляющая переменная инициализируется в другом месте программы. *Выражение3* может опускаться, если операторы приращения находятся в теле структуры **for** или если приращение вообще не требуется. Выражение приращения в структуре **for** действует подобно автономному оператору С в конце тела цикла **for**. Поэтому все нижеследующие выражения

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

в блоке приращения структуры **for** являются эквивалентными. Многие программирующие на С предпочитают форму **counter**++, так как приращение происходит после выполнения тела цикла. Поэтому постинкрементная форма представляется более естественной. Поскольку переменная, получающая здесь приращение в преинкрементной или постинкрементной форме, не входит ыражение, обе формы приращения приводят к одному и тому же результату. Две точки с запятой в структуре **for** являются обязательными.

Распространенная ошибка программирования 4.3

Использование запятых вместо точек с запятой в заголовке структуры **for**.

Распространенная ошибка программирования 4.4

Помещение точки с запятой сразу после заголовка структуры **for** делает тело этой структуры пустым оператором. Обычно это является логической ошибкой.

4.5. Структура for: замечания и рекомендации

1. Блоки инициализации, условия продолжения цикла и приращения могут содержать арифметические выражения. Например, в предположении, что **x = 2** и **y = 10**, оператор

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

   эквивалентен оператору

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

2. «Приращение» может быть отрицательным (в этом случае оно в действительности является декрементом, и значение счетчика цикла на самом деле уменьшается).

3. Если условие продолжения цикла является ложным изначально, то *тело цикла не выполняется. Вместо этого выполнение продолжается* с оператора, следующего за структурой **for**.

4. Управляющая переменная часто выводится на печать или используется при вычислениях в теле цикла, но это не является обязательным. Использование переменной для управления повторением при отсутствии ссылок на нее в теле цикла — довольно распространенный случай.

5. Блок-схемы структур **for** и **while** очень похожи. Например, на рис. 4.4 показана блок-схема оператора

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

   Из этой блок-схемы становится понятно, что инициализация происходит только один раз и что приращение происходит после выполнения оператора тела цикла. Обратите внимание, что (кроме кружков и стрелок) блок-схема содержит только символы прямоугольников и символ ромба. Снова вообразите себе, что программист имеет доступ к большому ящику пустых структур **for** — их столько, сколько может потребоваться программисту, чтобы путем их суперпозиции и вложения в другие управляющие структуры сформировать структурную реализацию потока управления алгоритма. Затем, опять же, прямоугольники и ромбы заполняются соответствующими алгоритму действиями и решениями.

Хороший стиль программирования 4.8

Хотя значение управляющей переменной может модифицироваться в теле цикла **for**, это может привести к труднообнаружимым ошибкам. Лучше его не изменять.

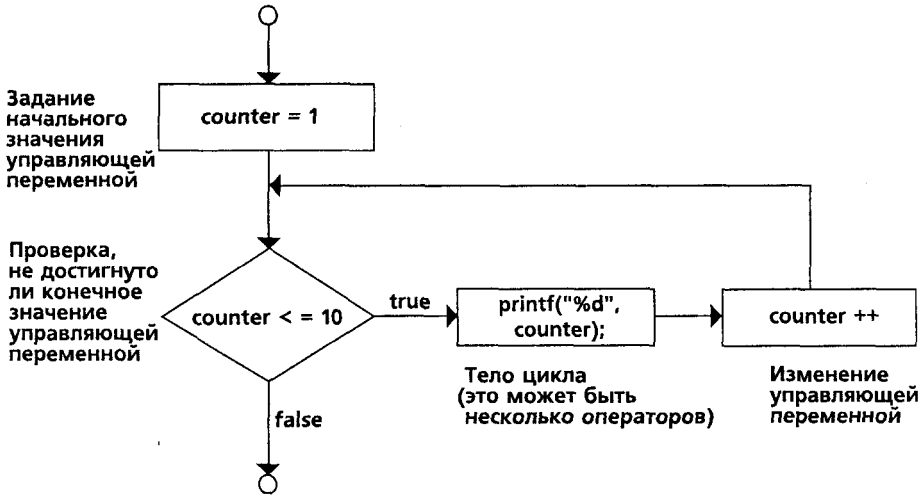


Рис. 4.4. Блок-схема типичной структуры **for**

4.6. Примеры структур for

Следующие примеры демонстрируют способы изменения управляющей переменной в структуре **for**.

a) Вариация управляющей переменной от **1** до **100** с приращением **1**.

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

b) Вариация управляющей переменной от **100** до **1** с приращением **-1** (декрементом **1**).

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

c) Вариация управляющей переменной от **7** до **77** с шагом, равным **7**.

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

d) Вариация управляющей переменной от **20** до **2** с шагом, равным **-2**.

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

e) Вариация управляющей переменной по следующей последовательности значений: **2, 5, 8, 11, 14, 17, 20**.

```
for (j = 2; j <= 20; j +=3)
```

f) Вариация управляющей переменной по следующей последовательности значений: **99, 88, 77, 66, 55, 44, 33, 22, 11, 0**.

```
for (j = 99; j >= 0; j -=11)
```

В следующих двух примерах даны простые приложения структуры **for**. Программа на рис. 4.5 использует структуру **for** для суммирования всех четных целых чисел от **2** до **100**.

Обратите внимание, что тело структуры **for** на рис. 4.5 на самом деле можно было бы соединить с правой частью заголовка структуры **for**, используя для этого операцию-запятую следующим образом:

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

Инициализацию **sum = 0** также можно было бы объединить с блоком инициализации структуры **for**.

```
/* Суммирование с помощью структуры for */
#include <stdio.h>

main()
{
   int sum = 0, number;

   for (number = 2; number <= 100; number += 2)
      sum += number;
   printf("Sum is %d\n", sum);

   return 0;
}
```

```
Sum is 2550
```

Рис. 4.5. Суммирование с помощью структуры **for**

Хороший стиль программирования 4.9

Хотя операторы, предшествующие структуре **for**, и операторы, содержащиеся в ее теле, часто могут быть объединены в заголовке структуры **for**, избегайте этого, поскольку это делает программу более трудной для чтения.

Хороший стиль программирования 4.10

Ограничивайте, если это возможно, размер заголовков управляющих структур одной строкой.

Во втором примере со структурой **for** вычисляются сложные проценты. Рассмотрим следующую постановку задачи:

Некий клиент открыл в банке 5-процентный сберегательный счет на $1000.00. Предполагая, что вся прибыль остается на депозите счета, рассчитайте и выведите сумму денег на счете на конец каждого года за 10 лет. Для определения этих сумм используйте следующую формулу:

$a = p(1 + r)^n$

где

p — первоначально вложенная сумма (т.е. сумма, на которую начисляются проценты)

r — годовая процентная ставка

n — число лет

a — сумма на депозите на конец n-го года.

Эта задача требует цикла, в котором выполняется указанный расчет баланса счета для каждого года 10-летнего периода. Решение задачи показано на рис. 4.6.

```
/ * Вычисление сложных процентов * /
#include <stdio.h>
#include <math.h>

main()
{
   int year;
   double amount, principal = 1000.0, rate = .05;

   printf("%4s%21s\n", "Year", "Amount on deposit");

   for (year = 1; year <= 10; year++) {
      amount = principal * pow(1.0 + rate, year);
      printf("%4d%21.2f\n", year, amount);
   }

   return 0;
}
```

```
Year    Amount on deposit
   1              1050.00
   2              1102.50
   3              1157.62
   4              1215.51
   5              1276.28
   6              1340.10
   7              1407.10
   8              1477.46
   9              1551.33
  10              1628.89
```

Рис. 4.6. Вычисление сложного процента с помощью **for**

Структура **for** выполняет 10 раз тело цикла, изменяя при этом управляющую переменную от 1 до 10 с приращением 1. Хотя в язык С не включена операция возведения в степень, мы можем, тем не менее, вызвать для этой цели стандартную библиотечную функцию **pow**. Функция **pow(x, y)** вычисляет значение **x** в степени **y**. Она принимает два параметра типа **double** и возвращает значение типа **double**. Тип **double** представляет собой тип с плавающей точкой, очень похожий на **float**, но в переменной типа **double** может храниться гораздо большее значение и с большей точностью, чем в переменной типа **float**. Обратите внимание, что всякий раз, когда используется математическая функция типа **pow**, должен быть включен заголовочный файл **math.h**. На самом деле без включения файла **math.h** эта программа работала бы неправильно. Функция **pow** требует двух параметров типа **double**. Обратите внимание, что переменная **year** является целым числом. Заголовочный файл **math.h** включает информацию, которая заставляет компилятор перед вызовом функции **pow** преобразовать значение переменной **year** во временное представление типа **double**. Эта информация содержится в так называемом прототипе функции **pow**. Прототипы функций являются важным нововведением языка ANSI С и обсуждаются в главе 5. Там же мы дадим краткое описание функции **pow** и других библиотечных математических функций.

Обратите внимание, что мы объявили переменные **amount**, **principal** и **rate** с типом **double**. Мы сделали это для простоты, поскольку имеем здесь дело с дробными долями доллара.

Хороший стиль программирования 4.11

Не используйте переменных типа **double** или **float** для проведения вычислений, связанных с денежными суммами. Неточность чисел с плавающей точкой может вызвать ошибки, которые приведут к неправильным значениям денежных сумм. В упражнениях мы изучим возможности использования целых чисел для проведения вычислений, связанных с денежными суммами.

Вот простое пояснение того, что может происходить при использовании типов **float** или **double** для представления долларовых значений.

Две денежные суммы, хранящиеся в машине в переменных типа **float**, могут быть равны 14.234 (выводится в виде 14.23 со спецификацией преобразования **%.2f**) и 18.673 (со спецификацией преобразования **%.2f** выводится в виде 18.67). При сложении этих значений получается сумма 32.907, которая со спецификацией преобразования **%.2f** выводится в виде 32.91. Таким образом, ваша распечатка может выглядеть как

```
  14.23
+ 18.67
-------
  32.91
```

но очевидно, что сумма отдельных чисел, в том виде, как они напечатаны, должна быть 32.90!

Для вывода значения переменной **amount** в программе используется спецификатор преобразования **%21.2f**. **Число 21** в спецификаторе обозначает *ширину поля*, в которое будет выводиться значение. Ширина поля **21** указывает на то, что выводимое значение займет **21** позицию вывода. Число **2** специфицирует точность (т.е. число позиций после десятичной точки). Если количество отображаемых символов меньше ширины поля, то значение будет автоматически *выровнено по правому краю*. Это особенно полезно для выравнивания значений с десятичной точкой, выводимых с одной и той же точностью. Чтобы выровнять значение *по левому краю*, поместите символ «-» (знак минус) между % и шириной поля. Обратите внимание, что знак минус также может применяться для выравнивания по левому краю целых чисел (как, например, в **%-6d**) и символьных строк (как, например, в **%-8s**). Мы подробно обсудим эти мощные средства форматирования функций **printf** и **scanf** в главе 9.

4.7. Структура со множественным выбором switch

В главе 3 мы обсуждали структуру с одним выбором **if** и структуру с двойным выбором **if/else**. Иногда алгоритм содержит последовательность принятия решений, когда происходит независимая проверка переменной или выражения на равенство каждому из константных целочисленных значений, которые они могут принимать, и в зависимости от этого предпринимаются различные действия. В языке C для обработки такого рода ситуаций, связанных с принятием решений, предусмотрена структура со множественным выбором **switch**.

Структура **switch** состоит из ряда меток **case** и необязательного блока **default**. В программе на рис. 4.7 структура **switch** используется для подсчета количества различных буквенных оценок, полученных студентами на экзамене.

```
/* Подсчет буквенных оценок */
#include <stdio.h>

main()
{
   int grade;
   int aCount = 0, bCount = 0, cCount = 0,
   dCount = 0, fCount = 0;

   printf("Enter the letter grades. \n");
   printf("Enter the EOF character to end input. \n");

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

      switch (grade) {          /* switch, вложенная в while */

         case 'A': case 'a':  /* grade равна A в верхнем */
            ++aCount;         /* или a в нижнем регистре */
            break;

         case 'B': case 'b':  /* grade равна B в верхнем */
            ++bCount;         /* или b в нижнем регистре */
            break;

         case 'C': case 'c':  /* grade равна C в верхнем */
            ++cCount;         /* или c в нижнем регистре */
            break;

         case 'D': case 'd':  /* grade равна D в верхнем */
            ++dCount;         /* или d в нижнем регистре */
            break;

         case 'F': case 'f':  /* grade равна F в верхнем */
            ++fCount;         /* или f в нижнем регистре */
            break;

         case '\n': case ' ': /* этот ввод игнорируется */
            break;

         default:             /* ловит все другие символы */
            printf("Incorrect letter grade entered.");
            printf(" Enter a new grade. \n");
            break;
      }
   }

   printf("\nTotals for each letter grade are:\n");
   printf("A: %d\n", aCount);
   printf("B: %d\n", bCount);
   printf("C: %d\n", cCount);
   printf("D: %d\n", dCount);
   printf("E: %d\n", eCount);
   printf("F: %d\n", fCount);

   return 0;
}
```

Рис. 4.7. Пример использования структуры **switch** (часть 1 из 2)

```
Enter the letter grades.
Enter the EOF character to end input.
A
B
C
C
A
D
F
C
E
Incorrect letter grade entered. Enter a new grade.
D
A
B

Totals for each letter grade are:
A: 3
B: 2
C: 3
D: 2
F: 1
```

Рис. 4.7. Пример использования структуры **switch** (часть 2 из 2)

В программе пользователь вводит буквенные оценки для группы. Внутри заголовка структуры **while**

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

сначала выполняется заключенное в круглые скобки присваивание **(grade = getchar())**. Функция **getchar** (из стандартной библиотеки ввода/вывода) считывает один символ с клавиатуры и сохраняет этот символ в целочисленной переменной **grade**. Символы обычно хранятся в переменных типа **char**. Однако важной особенностью языка C является то, что символы могут храниться в любом целочисленном типе данных, поскольку они представляются в компьютере в виде 1-байтовых целых чисел. Таким образом, мы можем обрабатывать символ либо как целое число, либо как символ в зависимости от его смысла. Например, оператор

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

использует спецификаторы преобразования **%c** и **%d** соответственно для вывода символа a и его целочисленного значения. Результатом будет

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

Целое число 97 является численным представлением символа в компьютере. Многие компьютеры в настоящее время используют набор символов ASCII (American Standard Code for Information Interchange), в котором число 97 представляет символ строчной буквы **'a'**. Список символов ASCII и их десятичных значений представлен в приложении Г. Символы можно считывать с помощью функции **scanf** со спецификатором преобразования **%c**.

Операторы присваивания, взятые целиком, на самом деле обладают значением. *Это в точности то самое значение, которое присваивается переменной* слева от знака =. Значением присваивания **grade = getchar()** является символ, возвращаемый функцией **getchar** и присваиваемый переменной **grade**.

Тот факт, что операторы присваивания обладают значением, может оказаться полезным для инициализации нескольких переменных одним и тем же значением. Например, в операторе

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

сначала оценивается присваивание **c = 0** (поскольку операция = ассоциируется справа налево). Затем переменной **b** присваивается значение присваивания **c = 0** (которое равно 0). Затем переменной a присваивается значение присваивания **b = (c = 0)** (которое также равно 0). В программе значение присваивания **grade = getchar()** сравнивается со значением **EOF** (символом, мнемоника которого означает «end of file» — «конец файла»). Мы принимаем **EOF** (который обычно имеет значение -1) в качестве контрольного значения. Пользователь вводит системно-зависимую комбинацию клавиш, означающую «конец файла», т.е. «У меня нет больше входных данных». **EOF** представляет собой символическую целочисленную константу, определенную в заголовочном файле **<stdio.h>** (мы увидим, как определяются символические константы, в главе 6). Если значение, присвоенное переменной **grade**, равно **EOF**, программа завершается. Мы решили представлять символы в этой программе в виде значений типа **int**, поскольку **EOF** имеет целочисленное значение (как мы уже говорили, обычно -1).

Совет по переносимости программ 4.1

Комбинации клавиш для ввода **EOF** (конца файла) являются системно-зависимыми.

Совет по переносимости программ 4.2

Проверка значения на равенство символической константе **EOF**, а не числу -1 делает программы более переносимыми. Стандарт ANSI требует, чтобы **EOF** являлась отрицательным целым числом (но не обязательно -1). Таким образом, на различных вычислительных системах **EOF** может иметь различные значения.

На системах UNIX и многих других индикатор **EOF** вводится нажатием *<return> <ctrl-d>*

Эта запись означает нажатие клавиши возврата каретки и затем одновременное нажатие клавиш **ctrl** и **d**. На других системах, например, VAX VMS корпорации DEC или MS-DOS корпорации Microsoft, индикатор **EOF** можно ввести при помощи

<ctrl-z>

Пользователь вводит с клавиатуры оценки. После нажатия клавиши возврата (Enter) функция **getchar** считывает одиночный символ. Если он не равен **EOF**, происходит вход в структуру **switch**. За ключевым словом **switch** следует имя переменной **grade** в круглых скобках. Она является здесь управляющим выражением. Значение этого выражения сравнивается с каждой из меток **case**. Предположим, пользователь ввел в качестве оценки символ **C**. Символ **C** автоматически сравнивается с каждой меткой case структуры **switch**. Если имеет место соответствие (**case 'C':**), выполняются операторы для этого **case**. В случае символа **C** переменная **cCount** увеличивается на **1** и по оператору **break** происходит немедленный выход из структуры **switch**.

Оператор **break** вызывает передачу управления первому оператору после структуры **switch**. Необходимость оператора **break** обусловлена тем, что без него метки **case** оператора **switch** могли бы выполняться подряд. Если **break** в операторе **switch** вообще отсутствует, то всякий раз, когда в структуре **switch**

имеет место соответствие, будут выполняться операторы для всех последующих меток **case**. (Эта особенность редко используется, хотя она идеально подходит для программирования итеративного стишка «Дом, который построил Джек».) Если не имеется ни одного соответствия, то выполняется блок **default** и выводится сообщение об ошибке.

Каждый блок **case** может включать одно или более действий. Структура **switch** отличается от всех других структур тем, что в блоке **case** структуры **switch** последовательность операторов не требуется заключать в фигурные скобки. Общая блок-схема структуры со множественным выбором **switch** (с операторами **break** в каждом блоке **case**) приводится на рис. 4.8.

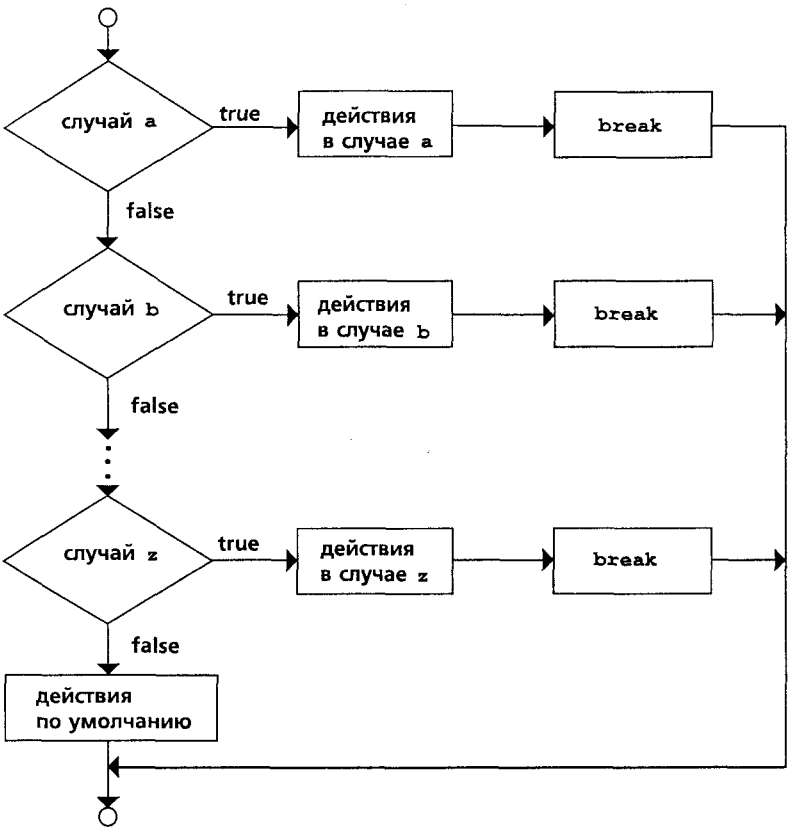


Рис. 4.8. Структура со множественным выбором **switch**

Из этой блок-схемы становится понятно, что каждый оператор **break** в конце блока **case** приводит к немедленной передаче управления за пределы структуры **switch**. Снова представьте себе, что программист имеет доступ к большому ящику пустых структур **switch** — их столько, сколько может потребоваться программисту, чтобы путем их суперпозиции и вложения в другие управляющие структуры сформировать структурную реализацию потока управления алгоритма. Затем прямоугольники и ромбы заполняются соответствующими алгоритму действиями и решениями.

Распространенная ошибка программирования 4.5

Пропуск оператора **break** в структуре **switch**, когда он необходим.

Хороший стиль программирования 4.12

Не забывайте о блоке **default** в операторах **switch**. Случаи, явно не проверяемые в структуре **switch**, игнорируются. Блок **default** помогает избежать этого, фокусируя внимание программиста на необходимости обрабатывать необычные условия. Существуют ситуации, когда никакой обработки по умолчанию (в блоке **default**) не требуется.

Хороший стиль программирования 4.13

Хотя предложения **case** и предложение **default** в структуре **switch** могут появляться в произвольном порядке, хорошим стилем программирования считается размещение предложения **default** в конце структуры.

Хороший стиль программирования 4.14

В том случае, когда в структуре **switch** предложение **default** указано последним, оператора **break** для него не требуется. Однако некоторые программисты вводят этот **break** для ясности и единообразия с метками **case**.

В структуре **switch** на рис. 4.7 строки

```
case '\n': case ' ':
   break;
```

заставляют программу пропускать символы новой строки и пробелы. Чтение символов по одному может вызывать некоторые проблемы. Чтобы заставить программу считывать символы, их нужно отправить компьютеру путем нажатия на клавиатуре клавиши возврата. Это приводит к помещению во входной поток символа новой строки после символа, который мы хотим обработать. Часто, чтобы программа работала правильно, этот символ новой строки должен обрабатываться особым образом. Включив в нашу структуру **switch** вышеприведенную метку **case**, мы предотвращаем вывод сообщения об ошибке в блоке **default**, когда во входном потоке встречаются символы новой строки или пробелы.

Распространенная ошибка программирования 4.6

Отсутствие обработки символов новой строки во входном потоке при чтении символов по одному может приводить к логическим ошибкам.

Хороший стиль программирования 4.15

Не забывайте о возможной необходимости обработки символов новой строки во входном потоке при считывании символов по одному.

Обратите внимание, что перечисление вместе нескольких меток case (как, например, **case 'D': case 'd':** на рис. 4.7) просто означает, что для каждой из этих меток должна быть выполнена одна и та же последовательность действий.

Применяя структуру **switch**, не забывайте, что она может быть использована только для проверки *константного целочисленного выражения*, т.е. лю-

бой комбинации символьных и целочисленных констант, которая раскрывается в константное целое значение. Символьная константа представляется в виде отдельного символа в одиночных кавычках, как, например 'A'. Чтобы быть распознанными в качестве символьных констант, символы должны быть заключены в одиночные кавычки. Целочисленные константы — это просто целые значения. В нашем примере мы использовали символьные константы. Не забывайте, что символы фактически представляют собой небольшие целые значения.

Переносимые языки программирования, подобные C, должны иметь гибкие размеры типов данных. Различным приложениям могут понадобиться целые числа различных размеров. Язык C предусматривает несколько типов данных для представления целых чисел. Диапазон целочисленных значений для каждого типа зависит от аппаратных средств конкретного компьютера. В дополнение к типам **int** и **char** в C предусмотрены типы **short** (сокращение от **short int**) и **long** (сокращение от **long int**). Стандарт ANSI определяет, что минимальный диапазон значений для целых чисел типа **short** равен ±32767. Целых чисел типа **long** достаточно для подавляющего большинства целочисленных расчетов. Стандарт определяет, что минимальный диапазон значений для целых чисел типа **long** равен ±2147483647. На большинстве компьютеров тип **int** эквивалентен типам **short** или **long**. Стандарт устанавливает, что диапазон значений для числа типа **int** по крайней мере совпадает с диапазоном значений целых чисел типа **short** и не превосходит диапазона значений целых чисел типа **long**. Тип данных **char** может применяться для представления целых чисел в диапазоне ±127 или любого символа из набора символов компьютера.

Совет по переносимости программ 4.3

Поскольку тип **int** различается по размеру от системы к системе, пользуйтесь целыми типа **long**, если вы рассчитываете обрабатывать целые числа вне диапазона ±32767 и хотели бы иметь возможность выполнять программу на различных компьютерных системах.

Совет по повышению эффективности 4.1

В случаях, ориентированных на эффективность выполнения, когда нужно экономить память или необходимо быстродействие, может оказаться предпочтительным использование целых чисел меньших размеров.

4.8. Структура повторения do/while

Структура повторения **do/while** подобна структуре **while**. В структуре **while** условие продолжения цикла проверяется в начале цикла до выполнения его тела. Структура **do/while** проверяет условие продолжения цикла после выполнения тела цикла, поэтому тело цикла будет выполнено по крайней мере один раз. После завершения цикла **do/while** выполнение программы продолжается с оператора, следующего за предложением **while**. Обратите внимание, что при наличии только одного оператора в теле цикла в структуре **do/while** нет необходимости использовать фигурные скобки. Однако обычно фигурные скобки включаются во избежание путаницы между структурами **do/while** и **while**. Например,

```
while (условие)
```

обычно рассматривается в качестве заголовка структуры **while**. Структура **do/while** без фигурных скобок вокруг ее тела, состоящего из одного оператора, выглядит как

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

что может приводить к путанице. Последняя строка — **while (*условие*);** может быть неправильно истолкована читателем как структура **while**, содержащая пустой оператор. Поэтому структура **do/while** с одним оператором часто записывается так:

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

Хороший стиль программирования 4.16

Некоторые программисты всегда включают фигурные скобки в структуру **do/while**, даже если они не являются необходимыми. Это помогает отличить структуру **do/while**, содержащую один оператор, от структуры **while**.

Распространенная ошибка программирования 4.7

Если условие продолжения цикла в структурах **while**, **for** или **do/while** никогда не становится ложным, возникают бесконечные циклы. Во избежание этого убедитесь, что сразу после заголовка структуры **for** или **while** нет точки с запятой. В цикле, управляемом счетчиком, убедитесь, что управляющая переменная в теле цикла увеличивается (или уменьшается). В цикле, управляемом контрольным значением, убедитесь в том, что оно рано или поздно вводится.

В программе на рис. 4.9 структура **do/while** используется для вывода чисел от 1 до 10. Обратите внимание, что к управляющей переменной **counter** при проверке условия продолжения цикла применяется операция преинкремента. Обратите также внимание на фигурные скобки, заключающие тело структуры **do/while** (состоящее из одного оператора).

```
/* Использование структуры повторения do/while */
# include <stdio.h>

main()
{
   int counter = 1;

   do {
      printf("%d  ", counter);
   } while (++counter <= 10);

   return 0;
}
```

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

Рис. 4.9. Использование структуры **do/while**

На рис. 4.10 представлена блок-схема структуры **do/while**. Из нее становится понятно, что условие продолжения цикла проверяется только после того, как действие будет выполнено по крайней мере один раз. Снова обратите внимание, что (кроме кружков и стрелок) блок-схема содержит только символ прямоугольника и символ ромба. Снова представьте себе, что программист имеет доступ к большому ящику пустых структур **do/while** — их столько, сколько может потребоваться программисту, чтобы путем их суперпозиции и вложения в другие управляющие структуры сформировать структурную реализацию потока управления алгоритма. Затем прямоугольники и ромбы заполняются соответствующими алгоритму действиями и решениями.

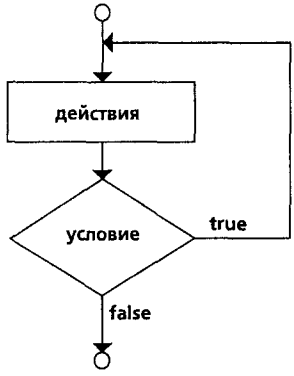


Рис. 4.10. Структура повторения **do/while**

4.9. Операторы break и continue

Операторы **break** и **continue** предназначены для изменения потока управления. Исполнение оператора **break** в структурах **while**, **for**, **do/while** или **switch** приводит к немедленному выходу из структуры. Выполнение программы продолжается с первого оператора, следующего за ней. Обычным применением оператора **break** является досрочный выход из цикла или пропуск оставшихся операторов в структуре **switch** (как на рис. 4.7). Рис. 4.11 демонстрирует оператор **break** в структуре повторения **for**. Когда структура **if** обнаруживает, что значение **x** стало равным **5**, выполняется оператор **break**. Это завершает оператор **for**, а программа продолжает выполняться с оператора **printf**, следующего за структурой **for**. Цикл выполняется полностью только четыре раза.

Выполнение оператора **continue** в структурах **while**, **for** или **do/while** приводит к пропуску оставшихся операторов в теле этой структуры и выполнению следующей итерации цикла. В структурах **while** и **do/while** проверка условия продолжения цикла проводится сразу же после выполнения оператора **continue**. В структуре **for** выполняется выражение приращения, а затем проверяется условие продолжения цикла. Ранее мы установили, что в большинстве случаев структура **for** может быть представлена структурой **while**. Одним из исключений является случай, когда выражение приращения в структуре **while** следует за оператором **continue**. В этом случае приращение

перед проверкой условия продолжения цикла не выполняется, и структура **while** работает не так, как структура **for**. На рис. 4.12 оператор **continue** служит в структуре **for** для пропуска оператора **printf** и начала следующей итерации цикла.

```
/* Использование оператора break в структуре for */
#include <stdio.h>

main()
{
   int x;

   for ( x = 1; x <= 10; x++)  {

      if (x == 5 )
         break;        /* прерывает цикл только если x == 5 */

      printf("%d ", x);
   }

   printf("\nBroke out of loop at x == %d\n", x);
   return 0;
}
```

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

Рис. 4.11. Использование оператора **break** в структуре **for**

```
/* Использование оператора continue в структуре for */
#include <stdio.h>

main ()
{
   int x;

   for ( x = 1; x <= 10; x++) {

      if ( x == 5 )
         continue;    /* пропускает оставшийся код в цикле только
                         если x == 5 */

      printf("%d ", x);
   }

   printf("\nUsed continue to skip printing the value 5\n");
   return 0;
}
```

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

Рис. 4.12. Использование оператора **continue** в структуре **for**

Хороший стиль программирования 4.17

Некоторым программистам кажется, что операторы **break** и **continue** нарушают нормы структурного программирования. Поскольку результаты действия этих операторов могут быть достигнуты структурными средствами (о которых мы скоро узнаем), эти программисты не пользуются операторами **break** и **continue**.

Совет по повышению эффективности 4.2

Операторы **break** и **continue**, при условии их надлежащего использования, выполняются быстрее соответствующих структурных методов, о которых мы скоро узнаем.

Общее методическое замечание 4.1

Существует определенное противоречие между разработкой качественного программного обеспечения и разработкой программного обеспечения, работающего наиболее эффективно. Часто одна из этих целей достигается за счет другой.

4.10. Логические операции

До сих пор мы изучали только простые условия, вроде **counter <= 10**, **total > 1000** и **number != sentinelValue**. Мы выражали эти условия с помощью операций отношений >, <, >= и <= и операций равенства == и !=. При каждом принятии решения проверялось в точности одно условие. Если бы в процессе принятия решения нам понадобилось проверить много условий, мы должны были бы делать эти проверки в отдельных операторах или во вложенных структурах **if** или **if/else**.

В языке C предусмотрены *логические операции*, посредством которых можно формировать сложные условия путем объединения более простых. Логическими операциями являются && (*логическое И*), (*логическое ИЛИ*) и ! (*логическое НЕ*, также называемое отрицанием). Мы на примерах рассмотрим каждую из этих операций.

Предположим, что в некотором месте программы мы хотим обеспечить истинность двух условий одновременно, прежде чем выберем определенную ветвь ее выполнения. В этом случае мы можем применить логическую операцию && следующим образом:

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

Этот оператор **if** содержит два простых условия. Условие **gender == 1** может оцениваться, например, для определения принадлежности определенного лица к женскому полу. Условие **age >= 65** оценивается для определения его принадлежности к пожилым гражданам. Сначала оцениваются два простых условия, поскольку приоритет обеих операций == и >= выше, чем приоритет операции &&. Затем в операторе **if** рассматривается объединенное условие

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

Это условие является истинным тогда и только тогда, когда оба простых условия истинны. И наконец, если это объединенное условие действительно становится истинным, то счетчик **seniorFemales** увеличивается на **1**. Если одно или оба простых условия ложны, то программа игнорирует приращение и переходит к оператору, следующему за **if**.

Таблица на рис. 4.13 сводит воедино свойства операции &&. В таблице показаны все четыре возможных комбинации нулевых (false) и ненулевых (true) значений для выражений expression1 и expression2. Такие таблицы часто называют таблицами истинности. В C все выражения, включающие операции отношений, операции равенства и/или логические операции, оцениваются как 0 или 1. Хотя C устанавливает для выражений значение true, равное 1, любое ненулевое значение воспринимается как истинное.

expression1	expression2	expression1 && expression2
0	0	0
0	ненулевое	0
ненулевое	0	0
ненулевое	ненулевое	1

Рис. 4.13. Таблица истинности операции && (логического И)

Теперь давайте рассмотрим операцию (логическое ИЛИ). Предположим, что в некотором месте программы мы хотим обеспечить истинность хотя бы одного из двух условий, чтобы перейти к некоторой ее ветви.

В этом случае мы применим операцию , как в следующем фрагменте программы:

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

Этот оператор также содержит два простых условия. Оценкой условия **semesterAverage >= 90** определяется, заслуживает ли студент оценки «A» за свои усердные занятия на протяжении семестра. Оценкой условия **finalExam >= 90** определяется, заслуживает ли студент оценки «A» за курс по причине выдающихся результатов на заключительном экзамене. После этого в операторе if рассматривается объединенное условие

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

и студент получает оценку «A», если одно или оба простых условия истинны. Обратите внимание, что сообщение **«Student grade is A (Оценка студента: A)»** не выводится только тогда, когда оба простых условия ложны (равны нулю). Рис. 4.14 показывает таблицу истинности для логической операции ИЛИ ( ).

expression1	expression2	expression1 expression2
0	0	0
0	ненулевое	0
ненулевое	0	0
ненулевое	ненулевое	1

Рис. 4.14. Таблица истинности операции (логического ИЛИ)

Операция && имеет более высокий приоритет, чем операция . Обе операции ассоциируются слева направо. Выражение, содержащее операции && или , оценивается только до тех пор, пока не установлена его истинность или ложность. Таким образом, оценка условия

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

прекращается, если переменная **gender** не равна **1** (т.е. выражение в целом ложно), и будет продолжена, если переменная **gender** равна **1** (т.е. выражение в целом все еще может быть истинным, если **age >= 65**).

Совет по повышению эффективности 4.3

В выражениях, включающих операцию &&, делайте условие, ложность которого наиболее вероятна, крайним слева. В выражениях, включающих операцию , делайте крайним слева условие, истинность которого наиболее вероятна. Это может уменьшить время выполнения программы.

В С предусмотрена операция ! (логическое отрицание), позволяющая программисту «обратить» смысл условия. В отличие от операций && и , которые объединяют два условия (и, следовательно, являются бинарными операциями), операция отрицания имеет в качестве операнда только одно условие (и, следовательно, является унарной). Логическая операция отрицания помещается перед условием, когда мы хотим выбрать ветвь выполнения программы с ложным (до отрицания) первоначальным условием, как, например, в следующем фрагменте программы:

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

Заключение условия **grade == sentinelValue** в круглые скобки необходимо, поскольку логическая операция отрицания имеет более высокий приоритет, чем операция равенства. Рис. 4.15 показывает таблицу истинности для логической операции отрицания.

expression	!expression
0	1
ненулевое	0

Рис. 4.15. Таблица истинности операции ! (логического отрицания)

В большинстве случаев программист может избежать использования логического отрицания, по-другому выражая условие с помощью соответствующей операции отношения. Например, предыдущий оператор может быть записан и так:

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

В таблице на рис. 4.16 показаны приоритет и ассоциативность различных операций языка С, рассмотренных к настоящему моменту. Операции показаны сверху вниз в порядке убывания их приоритета.

Операции	Ассоциативность	Тип
()	слева направо	круглые скобки
++ -- + - ! (тип)	справа налево	унарные
* / %	слева направо	мультипликативные
+ -	слева направо	адитивные
< <= > >=	слева направо	отношения
== !=	слева направо	равенства
&&	слева направо	логическое И
||	слева направо	логическое ИЛИ
?:	справа налево	условные
= += -= *= /= %=	справа налево	присваивания
,	слева направо	запятая

Рис. 4.16. Приоритет и ассоциативность операций

4.11. Смешивание операций равенства (==) и присваивания (=)

Есть одна типичная ошибка, которую программирующие на C, вне зависимости от их опыта, склонны делать настолько часто, что мы решили посвятить ей специальный раздел. Эта ошибка состоит в случайной замене операции == (равенства) на = (присваивания) или (реже) наоборот. Особый ущерб от этих замен связан с тем, что они обычно не вызывают синтаксических ошибок. Напротив, операторы с этими ошибками обычно компилируются правильно и программы выполняются до нормального завершения, возможно, порождая при этом неправильные результаты из-за логических ошибок времени выполнения.

Можно выделить два аспекта языка C, с которыми связаны эти проблемы. Во-первых, любое выражение в языке C, вырабатывающее значение, может быть использовано в блоке принятия решения любой управляющей структуры. Если это значение равно 0, оно обрабатывается как ложное, а если оно отлично от нуля, то обрабатывается как истинное. Второй аспект связан с тем, что в языке C присваивание возвращает значение, а именно то значение, которое присваивается переменной слева от знака =. Например, предположим, что мы намереваемся написать

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

но случайно мы пишем

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

Первый оператор **if**, как и положено, выдает премию сотруднику, платежный код которого равен 4. Второй оператор **if** — тот, что содержит ошибку, — вычисляет значение выражения присваивания в условии структуры **if**. Это выражение представляет собой простое присваивание, значением которого яв-

ляется константа 4. Поскольку любое ненулевое значение интерпретируется как «true», условие в этом операторе **if** всегда истинно и сотрудник всегда получает премию, независимо от его фактического платежного кода!

Распространенная ошибка программирования 4.8

Использование операции == для присваивания или использование операции = в качестве знака равенства.

Программисты обычно пишут в условиях типа **x == 7** имя переменной слева и константу справа от знака =. При обратном порядке записи — так, чтобы константа была слева, а имя переменной справа, как в выражении **7 == x**, — программист, случайно заменивший операцию == на операцию =, будет подстрахован компилятором. Компилятор обработает это как синтаксическую ошибку, поскольку слева от операции присваивания может помещаться только имя переменной. По крайней мере, это предотвратит возможные разрушительные последствия логической ошибки времени выполнения.

Говорят, что имена переменных являются *lvalue* (от «left value», «значение слева»), поскольку они могут стоять слева от операции присваивания. Константы, напротив, являются *rvalue* (от «right value», «значение справа»), поскольку они могут находиться только в правой части присваивания. Обратите внимание, что lvalue может использоваться также в качестве rvalue, но не наоборот.

Хороший стиль программирования 4.18

Когда в выражение равенства входят переменная и константа, как в случае **x == 1**, некоторые программисты предпочитают записывать в этом выражении константу слева, а имя переменной справа от знака = в качестве защитной меры от логической ошибки, возникающей при случайной замене программистом операции == операцией =.

Есть и другая сторона медали, которая может оказаться в равной степени неприятной. Предположим, что программист хочет присвоить значение переменной посредством простого оператора вроде

```
x = 1;
```

но вместо этого пишет

```
x == 1;
```

Здесь это также не является синтаксической ошибкой. Компилятор просто оценивает выражение как условие. Если переменная **x** равна **1**, условие истинно и выражение возвращает значение 1. Если переменная **x** не равна **1**, условие ложно и выражение возвращает значение 0. Вне зависимости от возвращаемого значения здесь отсутствует какая бы то ни было операция присваивания, поэтому это значение будет просто потеряно и значение x останется без изменений, вызывая, возможно, тем самым логическую ошибку времени выполнения. К сожалению, не существует удобного приема, чтобы помочь вам в разрешении этой проблемы!

4.12. Краткая сводка по структурному программированию

Подобно тому, как архитекторы проектируют различные сооружения, используя при этом коллективную мудрость представителей своей профессии, так же должны поступать и программисты при проектировании программ. Наша профессия моложе архитектуры и наша коллективная мудрость значительно беднее. Мы многому научились — всего лишь за пять десятилетий. Возможно, наиболее важным из того, что мы узнали, является то, что при помощи структурного программирования можно строить программы, которые проще для понимания (чем неструктурированные программы) и, следовательно, проще для тестирования, отладки, модификации и даже для проверки их корректности с точки зрения математики.

В главах 3 и 4 мы сосредоточились на управляющих структурах языка C. Каждая структура в отдельности была представлена вместе со своей блок-схемой и обсуждалась на примерах. Сейчас мы обобщим результаты глав 3 и 4 и представим простой набор правил для составления структурных программ и определения их свойств.

На рис. 4.17 приводится общая сводка по управляющим структурам, обсуждавшимся в главах 3 и 4. Для указания единственной точки входа в каждую структуру и единственной точки выхода из нее на рисунке используются символы кружков. Соединение отдельных символов блок-схемы в произвольном порядке может привести к неструктурированным программам. Поэтому программисты решили, объединяя символы блок-схемы, сформировать ограниченный набор управляющих структур и создавать только структурные программы путем надлежащего объединения управляющих структур только двумя простыми способами. Для простоты используются только структуры с одним входом/одним выходом — существует только один способ входа в каждую управляющую структуру и только один способ выхода из нее. Последовательное соединение управляющих структур для создания структурных программ несложно — точка выхода одной управляющей структуры непосредственно соединяется с точкой входа следующей управляющей структуры, т.е. управляющие структуры просто помещаются одна за другой в программе; мы назвали это «суперпозицией управляющих структур». Кроме того, правила построения структурированных программ допускают вложение управляющих структур.

На рис. 4.18 показаны правила построения правильно структурированных программ. Правила подразумевают, что символ прямоугольника может означать в блок-схеме любое действие, включая операции ввода/вывода.

Применение правил из рис. 4.18 всегда приводит к структурной блок-схеме с отчетливыми компоновочными блоками. Например, многократное применение правила 2 к простейшей блок-схеме приводит к структурной блок-схеме, содержащей набор последовательных прямоугольников (рис. 4.20). Обратите внимание, что по правилу 2 генерируется ряд расположенных друг за другом управляющих структур; давайте назовем правило 2 правилом суперпозиции.

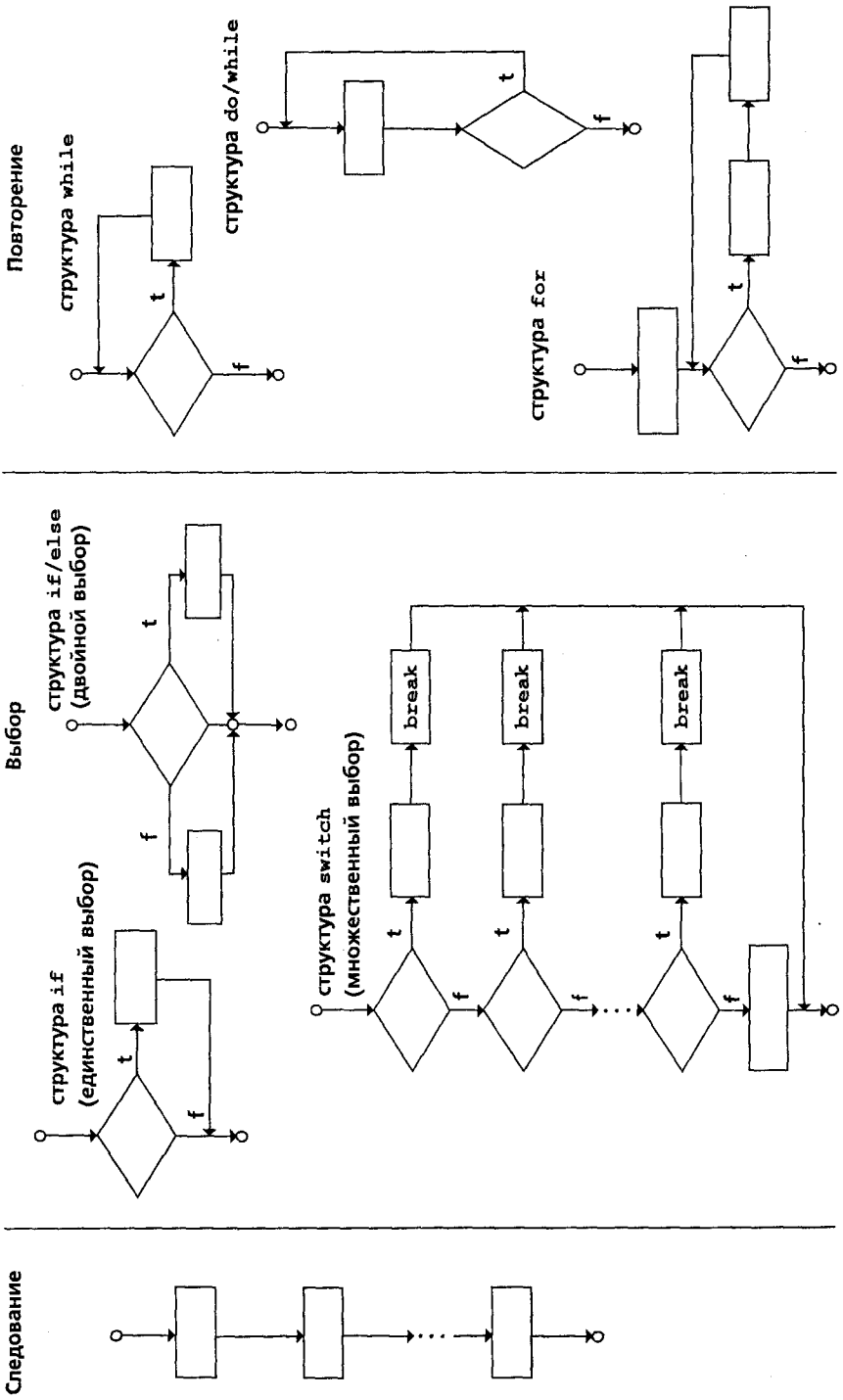


Рис. 4.17. Структуры языка C (последовательная, выбора и повторения) с одним входом/одним выходом

Правила построения структурированных программ
1) Начинайте с «простейшей блок-схемы» (рис. 4.19).
2) Любой прямоугольник (действие) может быть заменен двуя последовательными прямоугольниками (действиями).
3) Любой прямоугольник (действие) может быть заменен любой управляющей структурой (последовательной, **if**, **if/else**, **switch**, **while**, **do/while** или **for**.
4) Правила 2 и 3 могут применяться неограниченное количество раз в произвольном порядке.

Рис. 4.18. Правила построения структурных программ

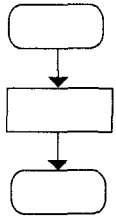


Рис. 4.19. Простейшая блок-схема

Правило 3 называется правилом вложения. Многократное применение правила 3 к простейшей блок-схеме приводит к блок-схеме с отчетливой вложенностью управляющих структур. Например, на рис. 4.21 прямоугольник в простейшей блок-схеме сначала заменяется структурой с двойным выбором **(if/else)**. Затем правило 3 снова применяется к обоим прямоугольникам в структуре с двойным выбором и происходит замена каждого прямоугольника своей структурой с двойным выбором. Изображенные штриховой линией прямоугольники вокруг каждой из структур с двойным выбором представляют исходный прямоугольник.

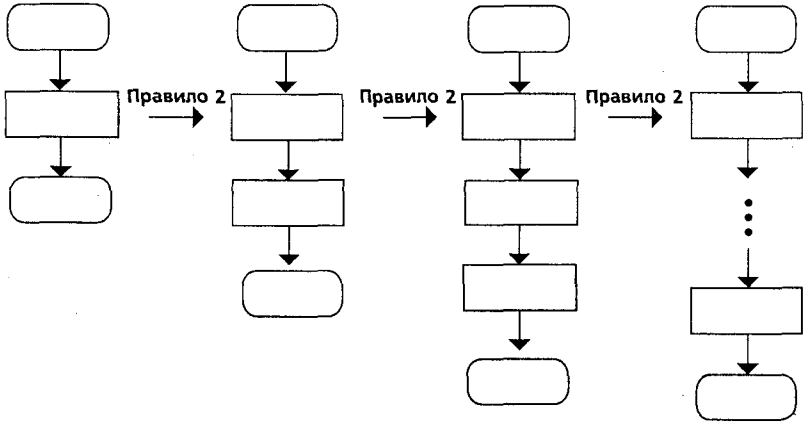


Рис. 4.20. Многократное применение правила 2 рис. 4.18 к простейшей блок-схеме

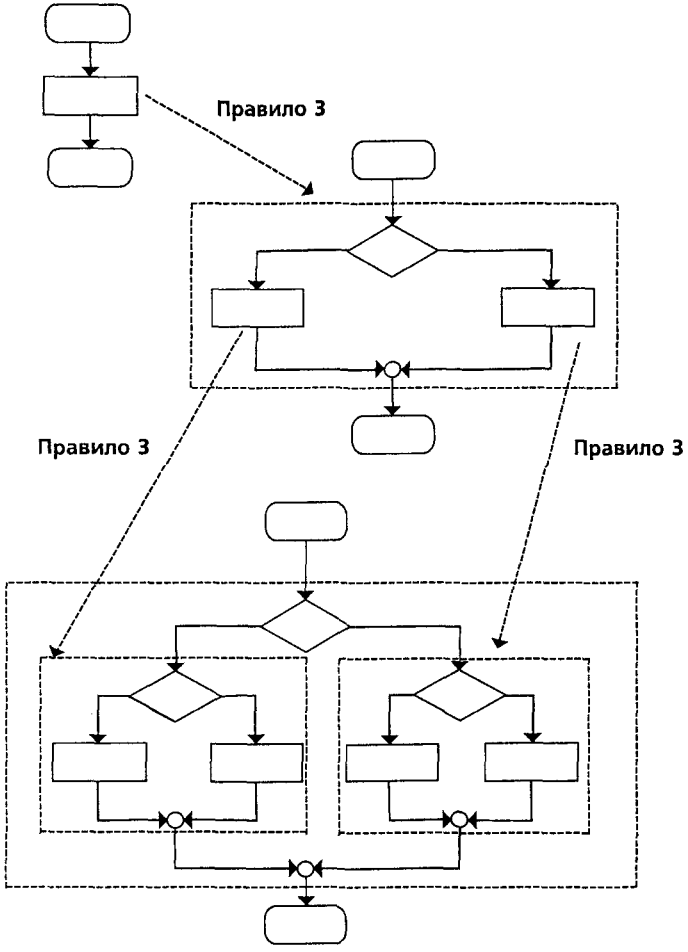


Рис. 4.21. Применение правила 3 рис. 4.18 к простейшей блок-схеме

Правило 4 порождает большие по размеру, более сложные структуры с более глубоким вложением. Блок-схемы, которые возникают в результате применения правил на рис. 4.18, составляют в совокупности набор всех возможных структурных блок-схем и, следовательно, набор всех возможных структурированных программ.

То, что их компоновочные блоки никогда не перекрываются, является следствием исключения оператора **goto**. Красота структурного подхода состоит в том, что мы ограничиваемся небольшим числом простых фрагментов с одним входом/одним выходом и компонуем их только одним из двух простых способов. На рис. 4.22 показаны виды суперпозиции компоновочных блоков, возникающих в результате применения правила 2, и виды вложенных компоновочных блоков, возникающих в результате применения правила 3. На этом рисунке также показаны перекрывающиеся компоновочные блоки, которые не могут появляться в структурированных блок-схемах (благодаря исключению оператора **goto**).

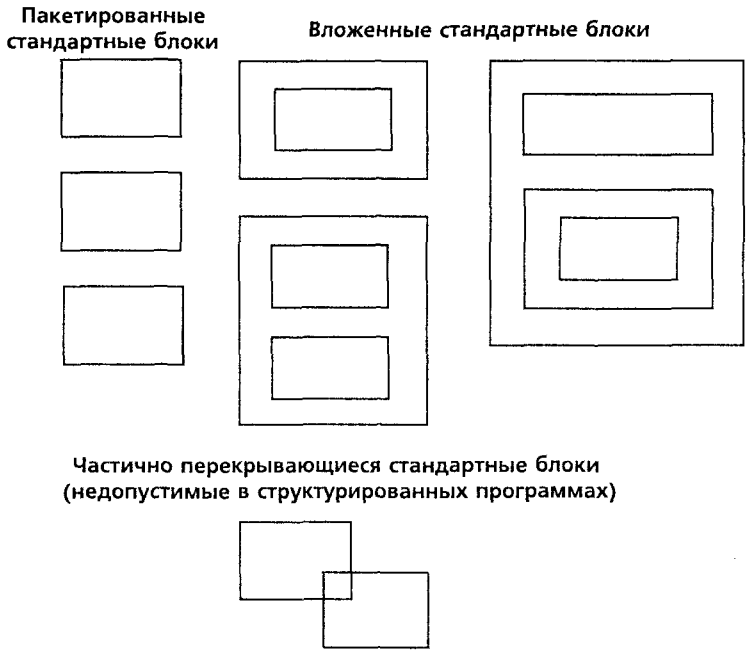


Рис. 4.22. Суперпозиция компоновочных блоков, вложенные компоновочные блоки и перекрывающиеся компоновочные блоки

При условии соблюдения правил на рис. 4.18 неструктурированная блок-схема (как, например, на рис. 4.23) получена быть не может. Если у вас есть сомнения в том, является ли данная блок-схема структурированной, применяйте правила рис. 4.18 в обратном порядке и попытайтесь свести ее к простейшей блок-схеме. Если блок-схему можно привести к простейшей, исходная блок-схема является структурированной; в противном случае она таковой не является.

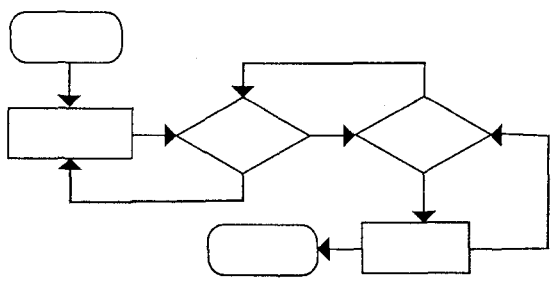


Рис. 4.23. Неструктурная блок-схема

Структурное программирование поощряет простоту. В результате исследований Бома и Якопини стало ясно, что необходимы только три разновидности программного управления:

- Последовательное
- Выбор
- Повторение

Последовательное управление тривиально. Выбор реализуется одним из трех способов:

- Структура **if** (один выбор)
- Структура **if/else** (двойной выбор)
- Структура **switch** (множественный выбор)

На самом деле несложно показать, что простой структуры **if** достаточно для обеспечения любой разновидности выбора — все, что можно сделать с помощью структур **if/else** и **switch**, может быть реализовано структурами **if**.

Повторение реализуется одним из трех способов:

- Структура **while**
- Структура **do/while**
- Структура **for**

Несложно показать, что структуры **while** достаточно для реализации любой разновидности повторения. Все, что можно сделать с помощью структур **do/while** и **for**, может быть сделано посредством структуры **while**.

Объединение этих результатов показывает, что любая разновидность программного управления, которая когда-либо может понадобиться в программе на C, может быть выражена при помощи всего лишь трех видов управления:

- последовательного
- структуры **if** (выбора)
- структуры **while** (повторения)

И эти управляющие структуры могут объединяться только двумя способами — суперпозицией и вложением. Воистину структурное программирование поощряет простоту.

В главах 3 и 4 мы обсуждали способы создания программ из управляющих структур, содержащих действия и решения. В главе 5 мы представим еще один элемент структурирования программ, называемый функцией. Мы научимся создавать большие программы, комбинируя функции, которые в свою очередь состоят из управляющих структур. Мы обсудим также, каким образом применение функций способствует многократному использованию программного кода.

Резюме

- Циклом называется группа команд, повторяемая компьютером многократно, пока не будет удовлетворено некоторое условие завершения. Двумя формами повторения являются: повторение, управляемое счетчиком, и повторение, управляемое контрольным значением.
- Счетчик цикла служит для подсчета количества повторений группы команд. Он увеличивается (обычно на 1) всякий раз, когда выполняется группа команд.
- Контрольные значения обычно применяются для управления в тех случаях, когда точное число повторений заранее не известно, и цикл со-

держит операторы, которые получают данные при каждом выполнении цикла.

- Контрольное значение вводится после того, как в программу были переданы все значимые элементы данных. Контрольные значения должны выбираться с осторожностью, — так, чтобы их нельзя было спутать с допустимыми элементами данных.
- Структура повторения **for** автоматически контролирует все детали повторения, управляемого счетчиком. Общий формат структуры **for**

```
for (выражение1; выражение2; выражение3)
   оператор
```

где *выражение1* инициализирует переменную управления циклом, *выражение2* является условием продолжения цикла и *выражение3* служит для приращения управляющей переменной.

- Структура повторения **do/while** подобна структуре **while**, однако в структуре **do/while** условие продолжения цикла проверяется в конце цикла, поэтому тело цикла будет выполнено по крайней мере один раз. Формат для оператора **do/while**

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

- Исполнение оператора **break** в какой-либо из структур повторения (**for**, **while** и **do/while**) приводит к немедленному выходу из структуры. Выполнение продолжается с первого оператора после цикла.
- Исполнение оператора **continue** в какой-либо из структур повторения (**for**, **while** и **do/while**) приводит к пропуску всех оставшихся операторов в теле структуры и переходу к следующей итерации цикла.
- Оператор **switch** обрабатывает последовательность решений, когда происходит проверка конкретной переменной или выражения на равенство каждому из значений, которые они могут принимать, и предпринимаются различные действия. Каждая метка **case** в операторе **switch** может вызывать выполнение нескольких операторов. В большинстве программ после операторов каждого блока **case** необходим оператор **break**, в противном случае программа будет выполнять операторы всех блоков **case** до тех пор, пока не будет встречен оператор **break** или достигнут конец оператора **switch**. В нескольких блоках **case** могут выполняться одни и те же операторы, если метки этих **case** будут перечислены вместе перед этими операторами. В структуре **switch** могут проверяться только константные целочисленные выражения.
- Функция **getchar** возвращает с клавиатуры (стандартный ввод) один символ в виде целого числа.
- На системах UNIX и многих других символ **EOF** вводится последовательностью

  <return> <ctrl-d>

  На системах VMS и DOS символ **EOF** вводится посредством

  <ctrl-z>
- Логические операции можно использовать для формирования сложных условий. Логическими операциями являются &&, и !, означающие

соответственно логическое И, логическое ИЛИ и логическое НЕТ (отрицание).

- Истинным значением является любое ненулевое значение.
- Ложным значением является 0 (нуль).

Терминология

break
char
continue
<ctrl-z>
double
long
lvalue («значение слева»)
<return><ctrl-d>
rvalue («значение справа»)
short
бесконечный цикл
блок **default** в структуре **switch**
вложенные управляющие структуры
выравнивание по левому краю
выравнивание по правому краю
декремент
знак минуса для выравнивания по левому краю
конец файла
конечное значение управляющей переменной
контрольное значение
логические операции
логическое И (&&)
логическое ИЛИ (||)
логическое отрицание (!)
метка case
множественный выбор
набор символов ASCII
начальное значение управляющей переменной
неопределенное повторение
определенное повторение
ошибка смещения счетчика
переменная управления циклом
повторение, управляемое счетчиком
правило вложения
правило суперпозиции
приращение управляющей переменной
простое условие
структура выбора **switch**
структура повторения **do/while**
структура повторения **for**
структура повторения **while**
структуры повторения
счетчик цикла
таблица истинности
тело цикла
унарная (одноместная) операция
управляющие структуры с одним входом / одним выходом
условие продолжения цикла
функция **getchar**
функция **pow**
ширина поля

Распространенные ошибки программирования

4.1. Ввиду того, что значения с плавающей точкой могут быть неточными, управление циклами со счетчиком при помощи переменных с плавающей точкой может привести к неточным значениям счетчика и ошибочным проверкам на окончание цикла.

4.2. Использование неправильной операции отношения или неправильного конечного значения счетчика цикла в условии структур **while** или **for** может привести к ошибке смещения счетчика.

4.3. Использование запятых вместо точек с запятой в заголовке структуры **for**.

4.4. Помещение точки с запятой сразу после заголовка структуры **for** делает тело этой структуры **for** пустым оператором. Обычно это является логической ошибкой.

4.5. Пропуск оператора **break** в структуре **switch**, когда он необходим.

4.6. Отсутствие обработки символов новой строки во входном потоке при чтении символов по одному может приводить к логическим ошибкам.

4.7. Если условие продолжения цикла в структурах **while**, **for** или **do/while** никогда не становится ложным, возникают бесконечные циклы. Во избежание этого убедитесь, что сразу после заголовка структуры **for** или **while** нет точки с запятой. В цикле, управляемом счетчиком, убедитесь, что управляющая переменная в теле цикла увеличивается (или уменьшается). В цикле, управляемом контрольным значением, убедитесь в том, что оно рано или поздно вводится.

4.8. Использование операции == для присваивания или использование операции = в качестве знака равенства.

Хороший стиль программирования

4.1. Управляйте циклами со счетчиком с помощью целочисленных значений.

4.2. Делайте отступы для операторов в теле каждой управляющей структуры.

4.3. Помещайте пустую строку до и после каждой управляющей структуры большого размера для выделения ее в тексте программы.

4.4. Слишком большое число уровней вложенности может сделать программу трудной для понимания. Пытайтесь избегать использования более трех уровней вложенности.

4.5. Сочетание междустрочного интервала до и после управляющих структур и отступов в теле управляющих структур в пределах их заголовков придает программам двумерный вид, который значительно повышает их удобочитаемость.

4.6. Использование конечного значения в условии структур **while** или **for** и операции отношения <= помогает избежать ошибок сдвига счетчика. Например, для цикла, используемого для вывода значений от 1 до 10, условием продолжения цикла должно быть **counter <= 10**, а не **counter < 11** или **counter < 10**.

4.7. Помещайте в разделы инициализации и приращения структуры **for** только выражения, включающие управляющие переменные. Манипуляции с другими переменными должны производиться либо перед циклом (если они выполняются только один раз, как операторы инициализации), либо в теле цикла (если они выполняются по одному разу за повторение, как операторы инкремента или декремента).

4.8. Хотя значение управляющей переменной может быть изменено в теле цикла **for**, это может привести к труднообнаружимым ошибкам; лучше его не изменять.

4.9. Хотя операторы, предшествующие структуре **for**, и операторы, содержащиеся в ее теле, часто могут быть объединены в заголовке структуры **for**, избегайте этого, поскольку это делает программу более трудной для чтения.

4.10. Ограничивайте, если это возможно, размер заголовков управляющих структур одной строкой.

4.11. Не используйте переменных типа **double** или **float** для проведения вычислений, связанных с денежными суммами. Неточность чисел с плавающей точкой может вызвать ошибки, которые приведут к неправильным значениям денежных сумм. В упражнениях мы изучим возможности использования целых чисел для проведения вычислений, связанных с денежными суммами.

4.12. Не забывайте о блоке **default** в операторах **switch**. Случаи, явно не проверяемые в структуре **switch**, игнорируется. Блок **default** помогает избежать этого, фокусируя внимание программиста на необходимости обрабатывать необычные условия. Существуют ситуации, когда никакой обработки по умолчанию (в блоке **default**) не требуется.

4.13. Хотя предложения **case** и предложение **default** в структуре **switch** могут появляться в произвольном порядке, хорошим стилем программирования считается размещение предложения **default** в конце структуры.

4.14. В том случае, когда в структуре **switch** предложение **default** указано последним, оператора **break** для него не требуется. Однако некоторые программисты вводят этот **break** для ясности и единообразия с метками **case**.

4.15. Не забывайте о возможной необходимости обработки символов новой строки во входном потоке при считывании символов по одному.

4.16. Некоторые программисты всегда включают фигурные скобки в структуру **do/while**, даже если они не являются необходимыми. Это помогает отличить структуру **do/while**, содержащую один оператор, от структуры **while**.

4.17. Некоторым программистам кажется, что операторы **break** и **continue** нарушают нормы структурного программирования. Поскольку результаты действия этих операторов могут быть достигнуты структурными средствами (о которых мы скоро узнаем), эти программисты не пользуются операторами **break** и **continue**.

4.18. Когда в выражение равенства входят переменная и константа, как в случае **x == 1**, некоторые программисты предпочитают записывать в этом выражении константу слева, а имя переменной справа от знака = в качестве защитной меры от логической ошибки, возникающей при случайной замене программистом операции == операцией =.

Советы по повышению эффективности

4.1. В случаях, ориентированных на эффективность выполнения, когда нужно экономить память или необходимо быстродействие, может оказаться предпочтительным использование целых чисел меньших размеров.

4.2. Операторы **break** и **continue**, при условии их надлежащего использования, выполняются быстрее соответствующих структурных методов, о которых мы скоро узнаем.

4.3. В выражениях, включающих операцию &&, делайте условие, ложность которого наиболее вероятна, крайним слева. В выражениях, включающих операцию , делайте условие крайним слева, истинность которого наиболее вероятна. Это может уменьшить время выполнения программы.

Советы по переносимости программ

4.1. Комбинации клавиш для ввода **EOF** (конца файла) являются системно-зависимыми.

4.2. Проверка значения на равенство символической константе **EOF**, а не числу -1 делает программы более переносимыми. Стандарт ANSI требует, чтобы **EOF** являлась отрицательным целым числом (но не обязательно -1). Таким образом, на различных вычислительных системах **EOF** может иметь различные значения.

4.3. Поскольку тип **int** различается по размеру от системы к системе, пользуйтесь целыми типа **long**, если вы рассчитываете обрабатывать целые числа вне диапазона ±32767 и хотели бы иметь возможность выполнять программу на нескольких различных компьютерных системах.

Общие методические замечания

4.1. Существует определенное противоречие между разработкой качественного программного обеспечения и разработкой программного обеспечения, работающего наиболее эффективно. Часто одна из этих целей достигается за счет другой.

Упражнения для самоконтроля

4.1. Заполните пробелы в каждом из следующих утверждений.

a) Повторение, управляемое счетчиком, также называется _____ повторением, поскольку заранее известно, сколько раз будет выполнен цикл.

b) Повторение, управляемое контрольным значением, также называется _____ повторением, поскольку заранее не известно, сколько раз будет выполнен цикл.

c) В повторении, управляемом счетчиком, для подсчета числа повторений группы команд используется _____.

d) Оператор _____ при его исполнении в структуре повторения вызывает немедленное выполнение следующей итерации цикла.

e) Оператор _____ при его исполнении в структуре повторения или структуре **switch** вызывает немедленный выход из структуры.

f) _____ используется для проверки данной переменной или выражения на равенство каждому из постоянных целочисленных значений, которые они могут принимать.

4.2. Установите, являются ли следующие высказывания верными или неверными. Если высказывание неверно, объясните почему.

a) В структуре выбора **switch** необходим блок **default**.

b) В блоке **default** структуры выбора **switch** необходим оператор **break**.

c) Выражение (**x > y && a < b**) истинно, если истинно либо выражение **x > y**, либо выражение **a < b**.

d) Выражение, содержащее операцию , истинно, если истинен один из или оба его операнда.

4.3. Напишите оператор C или последовательность операторов для решения каждой из следующих задач:

a) Просуммируйте нечетные целые числа от **1** до **99**, используя для этого структуру **for**. Предположите, что были объявлены целые переменные **sum** и **count**.

b) Выведите значение **333.546372** с шириной поля **15** символов с точностью **1**, **2**, **3**, **4** и **5**. Выровняйте выводимые значения по левому краю. Какими будут эти пять выводимых значений?

c) Вычислите значение **2.5** в степени **3**, используя для этого функцию **pow**. Выведите результат с точностью **2** и с шириной поля **10** позиций. Каким будет выводимое значение?

d) Выведите целые числа от 1 до 20, используя для этого цикл **while** и переменную-счетчик **x**. Предположите, что переменная **x** была объявлена, но не инициализирована. Выводите в строке только 5 целых чисел. Подсказка: используйте вычисление **x % 5**. Если это значение равно 0, выводите символ новой строки, в противном случае выводите символ табуляции.

e) Повторите упражнение 4.3 (d) с использованием структуры **for**.

4.4. Найдите ошибку в каждом из следующих фрагментов кода и объясните, как ее можно исправить.

a)
```
x = 1;
while (x <= 10);
   x++;
}
```

b)
```
for (y = .1; y != 1.0; y += .1)
   printf("%f\n", y);
```

c)
```
switch (n) {
   case 1:
      printf("The number is 1/n");
   case 2:
      printf("The number is 2/n");
      break;
   default:
      printf("The number is not 1 or 2\n");
      break;
}
```

d) Следующий код должен выводить значения от **1** до **10**.
```
n = 1;
while (n < 10)
   printf("%d ", n++);
```

Ответы к упражнениям для самоконтроля

4.1. a) определенным. b) неопределенным. c) контрольная переменная или счетчик. d) **continue**. e) **break**. f) структура выбора **switch**.

4.2. a) Неверно. Блок **default** не является обязательным. Если не требуется никакого действия по умолчанию, то нет необходимости и в блоке **default**.

b) Неверно. Оператор **break** используется для выхода из структуры **switch**. Оператор **break** не является необходимым, если блок **default** является последним блоком структуры.

c) Неверно. При использовании операции && для того, чтобы истинным было все выражение, должны быть истинными оба выражения отношений.

d) Верно.

4.3. a)
```
sum = 0;
for (count = 1; count <= 99; count += 2)
   sum += count;
```

b)
```
printf("%-15.1f\n", 333.546372);  /* выводит 333.5     */
printf("%-15.2f\n", 333.546372);  /* выводит 333.55    */
printf("%-15.3f\n", 333.546372);  /* выводит 333.546   */
printf("%-15.4f\n", 333.546372);  /* выводит 333.5464  */
printf("%-15.5f\n", 333.546372);  /* выводит 333.54637 */
```

c)
```
printf("%10.2f\n", pow(2.5, 3)); /* выводит 15.63 */
```

d)
```
x = 1;
while (x <= 20) {
   printf("%d", x);
   if (x % 5 == 0)
      printf("\n");
   else
      printf("\t");
   x++;
}
```

или

```
x = 1;
while (x <= 20)
   if (x % 5 == 0)
      printf("%d\n", x++);
   else
      printf("%d\t", x++);
```

или

```
x = 0;
while (++x <= 20)
   if (x % 5 == 0)
      printf("%d\n", x);
   else
      printf("%d\t", x);
```

e)
```
for (x = 1; x <= 20; x++) {
   printf("%d", x);
   if (x % 5 == 0)
      printf("%d\n");
```

```
      else
         printf("\t");
   }
```

или

```
   for (x = 1; x <= 20; x++)
      if (x % 5 == 0)
         printf("%d\n", x);
      else
         printf("%d\t", x);
```

4.4. a) Ошибка: помещение точки с запятой после заголовка структуры **while** приводит к бесконечному циклу.

Исправление: замените точку с запятой символом { или удалите ; и }.

b) Ошибка: использование числа с плавающей точкой для управления структурой повторения **for**.

Исправление: используйте целое число и выполните соответствующее вычисление, чтобы получить требуемые значения.

```
for (y = 1; y != 10; y++)
   printf("%f\n", (float) y / 10);
```

c) Ошибка: отсутствие оператора **break** в первом блоке **case**.

Исправление: добавьте оператор **break** в конец первого блока **case**. Обратите внимание, что это не обязательно является ошибкой, если программист хочет, чтобы оператор блока **case 2**: выполнялся всякий раз, когда выполняется оператор блока **case 1**:.

d) Ошибка: используется неправильная операция отношения в условии продолжения повторения структуры **while**.

Исправление: используйте операцию <= вместо <.

Упражнения

4.5. Найдите ошибку в каждом из следующих фрагментов кода (Примечание: может быть более одной ошибки):

a)
```
for (x = 100, x >= 1, x++)
   printf("%d\n", x);
```

b) Следующий код должен выводить, является ли данное целое число нечетным или нечетным:

```
switch (value % 2) {
   case 0:
      printf("Even integer\n");
   case 1:
      printf("Odd integer\n");
}
```

c) Следующий код должен вводить целое число и символ и выводить их. Предположите, что пользователь вводит с клавиатуры **100 A**.

```
scanf("%d", &intVal);
charval = getchar();
printf("Integer: %d\nCharacter: %c\n", intVal, charVal);
```

d)
```
for (x = .000001; x <= .0001; x += .000001)
   printf("%.7f\n", x);
```

e) Следующий код должен выводить нечетные целые числа от 999 до 1:

```
for (x = 999; x >= 1; x += 2)
   printf ("%d\n", x);
```

f) Следующий код должен выводить четные целые числа от 2 до 100:

```
counter = 2;
Do {
   if (counter % 2 == 0)
      printf("%d\n", counter);

   counter += 2;
} While (counter < 100);
```

g) Следующий код должен суммировать целые числа от 100 до 150 (предположите, что переменная total инициализирована значением 0):

```
for (x = 100; x <= 150; x++);
   total += x;
```

4.6. Установите, какие значения управляющей переменной x выводятся каждым из следующих операторов **for**:

a)
```
for(x = 2; x <= 13; x += 2)
   printf("%d\n", x);
```

b)
```
for(x = 5; x <= 22; x += 7)
   printf("%d\n", x);
```

c)
```
for (x = 3; x <= 15; x += 3)
   printf("%d\n", x);
```

d)
```
for (x = 1; x <= 5; x += 7)
   printf("%d\n", x)
```

e)
```
for (x = 12; x >= 2; x -= 3)
   printf("%d\n", x);
```

4.7. Напишите операторы **for**, которые выводят следующие последовательности значений:

a) 1, 2, 3, 4, 5, 6, 7

b) 3, 8, 13, 18, 23

c) 20, 14, 8, 2, -4, -10

d) 19, 27, 35, 43, 51

4.8. Что делает следующая программа?

```
#include <stdio.h>
main()
{
   int i, j, x, y;
   printf("Enter integers in the range 1-20: ")
   scanf("%d%d", &x, &y);
   for (i = 1; i <= y; i++) {
      for (j = 1; j <= x; j++)
         printf("@");
      printf("\n");
   }
   return 0;
}
```

4.9. Напишите программу, которая суммирует последовательность целых чисел. Предположите, что первое целое число, считываемое с помощью **scanf**, определяет количество значений, которое осталось ввести. Ваша программа должна считывать только одно значение при каждом выполнении **scanf**. Типичной входной последовательностью могло бы быть

```
5 100 200 300 400 500
```

где **5** указывает, что должны суммироваться последующие **5** значений.

4.10. Напишите программу, которая вычисляет и выводит среднее значение для нескольких целых чисел. Предположите, что последнем значением, считываемым с помощью **scanf**, является контрольное значение **9999**. Типичной входной последовательностью могла бы быть последовательность

```
10 8 11 7 9 9999
```

указывающая, что должно быть вычислено среднее для всех значений, предшествующих **9999**.

4.11. Напишите программу, которая находит наименьшее из нескольких целых чисел. Предположите, что первое считанное значение определяет количество оставшихся значений.

4.12. Напишите программу, которая вычисляет и выводит сумму четных целых чисел от 2 до 30.

4.13. Напишите программу, которая вычисляет и выводит произведение нечетных целых чисел от 1 до 15.

4.14. Функция факториала часто используется в задачах по теории вероятности. Факториал положительного целого числа n (записывается n! и произносится «n факториал») равен произведению положительных целых чисел от 1 до n. Напишите программу, которая вычисляет факториалы целых чисел от 1 до 5. Выведите результаты вычислений в табличной форме. Какие сложности могли бы помешать вам вычислить факториал числа 20?

4.15. Измените программу вычисления сложного процента из раздела 4.6 так, чтобы она повторяла свои шаги для процентных ставок 5 процентов, 6 процентов, 7 процентов, 8 процентов, 9 процентов и 10 процентов. Для варьирования процентной ставки используйте цикл **for**.

4.16. Напишите программу, которая по отдельности выводит один под другим следующие рисунки. Для генерации рисунков используйте циклы **for**. Все звездочки (*) должны выводиться единственным оператором **printf** в виде **printf("*");** (это вызывает вывод звездочек рядом друг с другом). Подсказка: для последних двух рисунков необходимо, чтобы каждая строка начиналась с соответствующего числа пробелов.

```
(A)          (B)          (C)          (D)
*            **********   **********            *
**           *********    *********           **
***          ********     ********           ***
****         *******      *******           ****
*****        ******       ******           *****
******       *****        *****           ******
*******      ****         ****           *******
********     ***          ***           ********
*********    **           **           *********
**********   *            *           **********
```

4.17. В период экономического спада востребование денежного долга значительно затрудняется, поэтому компании могут ограничивать предельные размеры кредитов своих клиентов, чтобы не допустить разрастания их годных к принятию счетов (денег, которые клиенты должны компании). В ответ на длительный спад некая компания урезала предельные размеры кредитов своих клиентов наполовину. Таким образом, если данный клиент имел предельный размер кредита в $2000, теперь предельный размер кредита этого клиента равен $1000. Если клиент имел предельный размер кредита в $5000, теперь предельный размер кредита этого клиента равен $2500. Напишите программу, которая анализирует состояние кредита трех клиентов этой компании. Для каждого клиента вам известны:

1. Номер счета клиента
2. Предельный размер кредита до спада
3. Текущий баланс клиента (т.е. сумма, которую клиент должен компании).

Ваша программа должна вычислить и вывести на печать новое значение предельного размера кредита для каждого клиента и определить (и распечатать) список тех клиентов, текущие балансы которых превышают их новое значение предельного размера кредита.

4.18. Одним интересным приложением компьютеров является рисование графиков и столбцовых диаграмм (иногда называемых «гистограммами»). Напишите программу, которая считывает пять чисел (каждое между 1 и 30). Для каждого считанного числа ваша программа должна вывести строку из равного этому числу количества смежных звездочек. Например, если ваша программа считывает число семь, она должна вывести *******.

4.19. Фирма, занимающаяся заказами по почте, продает пять различных видов изделий, розничные цены на которые показаны в следующей таблице:

Номер изделия	Розничная цена
1	$2.98
2	$4.50
3	$9.98
4	$4.49
5	$6.87

Напишите программу, которая считывает последовательность пар чисел следующим образом:

1. Номер изделия
2. Количество, проданное за один день

Ваша программа должна использовать оператор **switch**, чтобы определить розничную цену за каждое изделие. Программа должна вычислить и отобразить на экране итоговую сумму по розничной продаже всех изделий за последнюю неделю.

4.20. Закончите следующие таблицы истинности, заполняя каждый пробел 0 или 1.

Условие1	Условие2	Условие1 && Условие2
0	0	0
0	ненулевое	0
ненулевое	0	_____
ненулевое	ненулевое	_____

Условие1	Условие2	Условие1 Условие2
0	0	0
0	ненулевое	1
ненулевое	0	_____
ненулевое	ненулевое	_____

Условие1	!Условие1
0	1
ненулевое	_____

4.21. Перепишите программу на рис. 4.2 так, чтобы инициализация переменной **counter** выполнялась при объявлении, а не в структуре **for**.

4.22. Измените программу на рис. 4.7 так, чтобы она вычисляла среднюю оценку для группы.

4.23. Измените программу на рис. 4.6 так, чтобы для вычисления сложных процентов она использовала только целые числа. (Подсказка: обрабатывайте все денежные суммы в виде целочисленных значений центов. Затем «разбейте» результат на доллары и центы, используя для этого соответственно операции деления и взятия модуля. Вставьте десятичную точку.)

4.24. Предположим, что i = 1, j = 2, k = 3 и m = 2. Что выводит каждый из следующих операторов?

a) `printf("%d", i == 1);`

b) `printf("%d", j == 3);`

c) `printf("%d", i >= 1 && j < 4);`

d) `printf("%d", m <= 99 && k < m);`

e) `printf("%d", j >= i || k == m);`

f) `printf("%d", k + m < j || 3 - j >= k);`

g) `printf("%d", !m);`

h) `printf("%d", !(j - m));`

i) `printf("%d", !(k > m));`

j) `printf("%d", !(j > k));`

4.25. Выведите таблицу, состоящую из десятичного, двоичного, восьмеричного и шестнадцатеричного эквивалентов. Если вы не знакомы с этими системами счисления, но хотите попытаться выполнить это упражнение, сначала прочитайте приложение Д.

4.26. Вычислите значение π с помощью бесконечного ряда

$$\pi = 4 - \frac{4}{3} + \frac{4}{5} - \frac{4}{7} + \frac{4}{9} - \frac{4}{11} + \text{K}$$

Выведите таблицу, которая показывает значение π, аппроксимированное одним членом этого ряда, двумя членами, тремя членами и т.д. Сколько членов этого ряда нужно использовать, прежде чем вы впервые получите 3.14? 3.141? 3.1415? 3.14159?

4.27. *(Пифагоровы тройки.)* Прямоугольный треугольник может иметь стороны, каждая из которых является целым числом. Набор трех целочисленных значений для сторон прямоугольного треугольника называется пифагоровой тройкой. Эти три стороны должны удовлетворять следующему соотношению: сумма квадратов двух катетов равна квадрату гипотенузы. Найдите все пифагоровы тройки для катетов *side1*, *side2* и гипотенузы, не превосходящих 500. Используйте цикл **for** с тройной вложенностью, в котором просто перебираются все возможности. Это является примером вычисления «с помощью грубой силы». Многим оно не приносит эстетического удовлетворения. Но есть много причин, по которым эти методы важны. Во-первых, при мощности вычислительной техники, возрастающей такими необыкновенными темпами, решения, для получения которых понадобились бы годы или даже столетия компьютерного времени при использовании технологий, применявшихся всего лишь несколько лет тому назад, теперь могут быть получены за часы, минуты или даже секунды. Современные микропроцессорные схемы могут обрабатывать более 100 миллионов операций в секунду! И в 90-е годы, по всей вероятности, должны появиться микропроцессорные схемы, способные обрабатывать миллиард операций в секунду. Во-вторых, как вы узнаете из курсов по информатике для продолжающих обучение, существует большое число интересных задач, для которых не известны алгоритмические подходы, отличные от решения «с помощью грубой силы». В этой книге мы исследуем многие разновидности методологий решения задач. Мы рассмотрим многие подходы к различным интересным проблемам, связанные с решением «с помощью грубой силы».

4.28. Компания платит своим служащим как администраторам (это те, кто получает еженедельно фиксированную зарплату), работникам с почасовой оплатой (те, кто получает фиксированную почасовую оплату за первые 40 отработанных часов и «полуторную», т.е. пре-

вышающую в 1.5 раз их почасовую, оплату за сверхурочные часы работы), работникам с комиссионным вознаграждением (те, кто получает $250 плюс 5.7% от их валовых еженедельных продаж) и работникам со сдельной оплатой труда (те, кто получает фиксированную сумму денег за каждую единицу произведенной ими продукции — каждый сдельщик в этой компании работает только с одной разновидностью продукции). Напишите программу для расчета еженедельной зарплаты каждого служащего. Вам неизвестно заранее количество служащих. Служащий каждой из групп имеет собственный код оплаты: администраторы имеют код оплаты 1, работники с почасовой оплатой имеют код 2, работники с комиссионным вознаграждением имеют код 3 и работники со сдельной оплатой труда имеют код 4. Используйте оператор **switch** для расчета зарплаты каждого служащего в зависимости от его кода. Внутри оператора **switch** *попросите пользователя* (т.е. *клерка*, *рассчитывающего зарплату*) ввести соответствующие данные, которые потребуются вашей программе для расчета зарплаты каждого служащего, исходя из его кода оплаты.

4.29. *(Законы де Моргана.)* В этой главе мы обсуждали логические операции &&, и !. Законы де Моргана иногда могут помочь нам записать логическое выражение в более удобной форме. Эти законы устанавливают, что выражение !(*условие1* && *условие2*) логически эквивалентно выражению (!*условие1* !*условие2*). Также выражение !(*условие1* *условие2*) логически эквивалентно выражению (!*условие1* && !*условие2*). Используйте законы де Моргана для написания эквивалентного выражения для каждого из следующих выражений, а затем напишите программу, показывающую, что исходное и новое выражения в каждом случае являются эквивалентными:

a) `!(x < 5) && !(y >= 7)`

b) `!(a == b) || !(g != 5)`

c) `!((x <= 8) && (Y > 4))`

d) `!(i > 4) || (j <= 6))`

4.30. Перепишите программу на рис. 4.7, заменив оператор **switch** вложенным оператором **if/else**; будьте внимательны и надлежащим образом рассмотрите блок **default**. Затем перепишите эту новую версию, заменяя вложенные операторы **if/else** серией операторов **if**; здесь также будьте внимательны и рассмотрите реализацию блока **default** (это сделать труднее, чем в варианте с вложенным **if/else**). Упражнение показывает, что оператор **switch** введен в сущности для удобства и что любой такой оператор может быть записан только с помощью операторов с одним выбором.

4.31. Напишите программу, которая выводит следующую фигуру в виде ромба. Вы можете использовать операторы **printf**, которые выводят либо одну звездочку (*), либо один пробел. Максимизируйте количество используемых вами повторений (посредством вложенных структур **for**) и минимизируйте число операторов **printf**.

```
   *
  ***
 *****
*******
*********
*******
 *****
  ***
   *
```

4.32. Измените программу, которую вы написали в упражнении 4.31 так, чтобы она считывала нечетное число в диапазоне от 1 до 19 для задания количества строк в ромбе. Затем ваша программа должна отображать на экране ромб соответствующего размера.

4.33. Если вы знакомы с римскими цифрами, напишите программу, которая выводит таблицу всех римских эквивалентов десятичных чисел в диапазоне от 1 до 100.

4.34. Напишите программу, которая выводит таблицу двоичных, восьмеричных и шестнадцатеричных эквивалентов десятичных чисел в диапазоне от 1 до 256. Если вы не знакомы с этими системами счисления и хотели бы попытаться сделать это упражнение, сначала прочитайте приложение Д.

4.35. Опишите процесс, который потребовался бы для замены цикла **do/while** эквивалентным ему циклом **while**. Какая возникает проблема, когда вы пытаетесь заменить цикл **while** эквивалентным циклом **do/while**? Предположим, что вам сказали, что нужно убрать цикл **while** и заменить его циклом **do/while**. Какая дополнительная управляющая структура могла бы вам понадобиться и каким образом вы бы ее использовали, чтобы гарантировать, что получившаяся программа ведет себя точно так же, как оригинал.

4.36. Напишите программу, которая вводит год в диапазоне от 1994 до 1999 и использует повторение с циклом **for** для печати сжатого, аккуратного календаря. Обратите внимание на високосные годы.

4.37. Операторы **break** и **continue** критикуют за то, что каждый из них является неструктурным. В самом деле, операторы **break** и **continue** всегда можно заменить структурными операторами, хотя выглядеть это может неуклюже. Опишите в общем случае, каким образом вы могли бы убрать из цикла любой оператор **break** и заменить его некоторым структурным эквивалентом. (Подсказка: при помощи оператора **break** мы выходим из цикла, находясь внутри тела цикла. Другим способом выхода является отрицательный результат проверки на продолжение цикла. Рассмотрите использование при проверке на продолжение цикла второй проверки, которая указывает на «досрочный выход из-за условия 'break'».) Примените разработанный здесь метод для устранения оператора **break** из программы на рис. 4.11.

4.38. Что делает следующий фрагмент программы?

```
for (i = 1; j <= 5; i++) {
   for (j = 1; j <= 3; j++) {
      for (k = 1, k <= 4; k++)
         printf ("*");
      printf ("\n");
   }
   printf ("\n");
}
```

4.39. Опишите в общем случае, каким образом вы могли бы убрать из цикла любой оператор **continue** и заменить его некоторым структурным эквивалентом. Примените разработанный здесь метод для устранения оператора **continue** из программы на рис. 4.12.

4.40. Опишите в общем случае, каким образом вы могли бы убрать операторы **break** из структуры **switch** и заменить их структурными эквивалентами. Примените разработанный здесь метод (возможно неуклюжий) для устранения операторов **break** из программы на рис. 4.7.

Функции



Цели

- Понять принципы модульного построения программ из небольших блоков, называемых функциями.
- Познакомиться с часто используемыми математическими функциями стандартной библиотеки С.
- Научиться созданию новых функций.
- Понять механизмы обмена информацией между функциями.
- Познакомиться с методиками моделирования, основанными на генерации случайных чисел.
- Понять принципы написания и использования функций, вызывающих самих себя.

Содержание

5.1. Введение
5.2. Программные модули в C
5.3. Функции математической библиотеки
5.4. Функции
5.5. Определения функций
5.6. Прототипы функций
5.7. Заголовочные файлы
5.8. Вызов функций: вызов по значению и по ссылке
5.9. Генерация случайных чисел
5.10. Пример: стохастическая игра
5.11. Классы памяти
5.12. Правила области действия
5.13. Рекурсия
5.14. Пример применения рекурсии: числа Фибоначчи
5.15. Рекурсия в сравнении с итерацией

Резюме • Распространенные ошибки программирования • Хороший стиль программирования • Советы по переносимости программ • Советы по повышению эффективности • Общие методические замечания • Упражнения для самоконтроля • Ответы на упражнения для самоконтроля • Упражнения

5.1. Введение

Большинство компьютерных программ, которые решают задачи реального мира, намного больше тех, что представлены в первых главах этой книги. Как показывает практика, наилучшим способом разработки и поддержки больших программ является конструирование программы из небольших частей, или *модулей*, с каждым из которых обращаться проще, чем с первоначальной программой. Эта методика следует принципу «разделяй и властвуй». Данная глава описывает возможности языка C, которые упрощают проектирование, реализацию, использование и сопровождение больших программ.

5.2. Программные модули в C

Модули в C называются *функциями*. Программы на C обычно пишутся путем соединения новых функций, созданных программистом, с функциями, которые поставляются в составе *стандартной библиотеки C*. В этой главе мы обсудим оба вида функций. Стандартная библиотека C предоставляет широкий набор функций для выполнения общих математических вычислений, обработки строк и символов, ввода/вывода и многих других полезных операций. Стандартные функции упрощают работу программиста, поскольку удовлетворяют многим из его потребностей.

Хороший стиль программирования 5.1

Освойтесь с широким набором функций стандартной библиотеки ANSI C.

Общее методическое замечание 5.1

Не изобретайте колесо. Если есть возможность, используйте функции стандартной библиотеки ANSI C вместо того, чтобы создавать новые функции. Это уменьшает время разработки программы.

Совет по переносимости программ 5.1

Использование функций стандартной библиотеки ANSI C делает программы более переносимыми.

Хотя функции стандартной библиотеки технически не являются частью языка C, они неизменно поставляются с системами ANSI C. Функции **printf**, **scanf** и **pow**, которые мы использовали в предыдущих главах, являются функциями стандартной библиотеки.

Программист может написать функции для выделения конкретных задач, которые могут вызываться из многих точек программы. Эти функции иногда упоминаются как *функции, определенные программистом*. Операторы, определяющие функцию, пишутся только один раз, и эти операторы скрыты от других функций.

Обращение к функции осуществляется посредством *вызова функции*. В вызове функции указывается ее имя и передается информация (в качестве аргументов), необходимая функции для выполнения своей задачи. Аналогом такой процедуры служит иерархическая форма управления. Начальник (*вызывающая функция*) просит работника (*вызываемую функцию*) выполнить задание и, когда оно будет выполнено, сообщить об этом. Например, функция, которой требуется вывести информацию на экран, вызывает для выполнения задачи рабочую функцию **printf**; **printf** выводит информацию и передает (т.е. *возвращает*) вызывающей функции сообщение о выполнении своей задачи. Функция-начальник не знает, каким образом функция-работник выполняет поставленную задачу. Работник может вызывать другие рабочие функции, а начальник не будет об этом знать. Мы скоро увидим, как это «сокрытие» подробностей выполнения задачи способствует разработке хороших программ. На рис. 5.1 показана функция **main**, поддерживающая связь с несколькими рабочими функциями согласно законам иерархии. Обратите внимание, что **worker1** действует как функция-начальник по отношению к **worker4** и **worker5**. Отношения между функциями могут иметь и другую структуру, отличную от иерархической, показанной на этом рисунке.

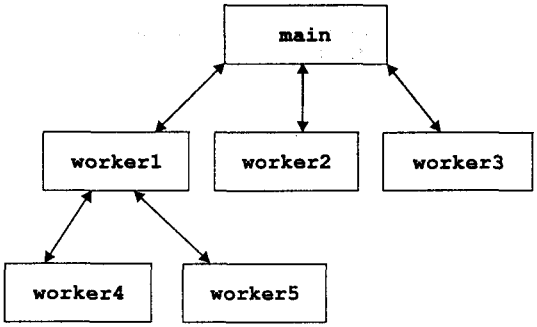


Рис. 5.1. Иерархия отношений функции-начальника и функции-работника

5.3. Функции математической библиотеки

Функции математической библиотеки позволяют программисту выполнять некоторые общие математические вычисления. Мы воспользуемся различными функциями математической библиотеки, чтобы познакомиться с концепцией функций. Позже в книге мы обсудим многие другие функции стандартной библиотеки С. Полный список функций стандартной библиотеки приведен в приложении Б.

Обычно к функции обращаются, записывая имя с последующими левой круглой скобкой, *аргументом* функции (или списком аргументов, отделяемых друг от друга запятыми) и правой круглой скобкой. Например, программист, желающий вычислить и напечатать значение квадратного корня из **900.0**, мог бы написать

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

Когда этот оператор выполняется, для вычисления значения квадратного корня из числа в скобках **(900.0)** производится вызов функции **sqrt** из математической библиотеки. Число **900.0** является аргументом функции **sqrt**. Показанный оператор напечатал бы **30.00**. Функция **sqrt** получает аргумент типа **double** и возвращает результат типа **double**. Все функции математической библиотеки возвращают данные типа **double**.

Хороший стиль программирования 5.2

При использовании функций математической библиотеки включите в программу заголовочный файл математики с помощью директивы препроцессора **#include <math.h>**.

Распространенная ошибка программирования 5.1

При отсутствии в программе заголовочного файла математики обращение программы к функциям математической библиотеки может приводить к странным результатам.

Аргументы функций могут быть константами, переменными или выражениями. Если **c1 = 13.0**, **d = 3.0** и **f = 4.0**, то оператор

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

вычисляет и печатает значение квадратного корня из **13.0+3.0*4.0=25.0**, т.е. **5.00**.

Некоторые функции математической библиотеки C приведены на рис. 5.2. Переменные **x** и **y** имеют тип **double**.

Функция	Назначение	Пример
sqrt(x)	квадратный корень из x	sqrt(900.0) равен 30.0 sqrt(9.0) равен 3.0
exp(x)	экспоненциальная функция e^x	exp(1.0) равна 2.718282 exp(2.0) равна 7.389056
log(x)	натуральный логарифм x (основание e)	log(2.718282) равен 1.0 log(7.389056) равен 2.0
log10(x)	логарифм x (основание 10)	log10(1.0) равен 0.0 log10(10.0) равен 1.0 log10(100.0) равен 2.0
fabs(x)	*абсолютное значение* x	*если* x > 0, *то* fabs(x) равно x если x=0, то fabs(x) равно 0.0 если x < 0, то fabs(x) равно -x
ceil(x)	округляет x до ближайшего целого, не меньшего x	ceil(9.2) равно 10.0 ceil(-9.8) равно -9.0
floor(x)	округляет x до ближайшего целого, не превосходящего x	floor(9.2) равно 9.0 floor(-9.8) равно -10.0
pow(x, y)	x возводится в степень y (x^y)	pow(2, 7) равно 128.0 pow(9, .5) равно 3.0
fmod(x, y)	остаток x/y как число с плавающей точкой	fmod(13.657, 2.333) равен 1.992
sin(x)	тригонометрический синус x (x в радианах)	sin(0.0) равен 0.0
cos(x)	тригонометрический косинус x (x в радианах)	cos(0.0) равен 1.0
tan(x)	тригонометрический тангенс x (x в радианах)	tan(0.0) равен 0.0

Рис. 5.2. Функции общего назначения математической библиотеки

5.4. Функции

Функции *позволяют программисту разбить программу на модули*. Все переменные, объявленные в определениях функций, являются *локальными переменными* — они известны только той функции, в которой определены. Большинство функций имеют список *параметров*. Параметры позволяют функциям обмениваться информацией. Параметры функции — это также локальные переменные.

Общее методическое замечание 5.2

В программах, содержащих большое число функций, **main** должна быть реализована как группа обращений к функциям, выполняющим основную часть работы.

Существует несколько оснований для разбиения программы на функции. Подход «разделяй и властвуй» делает разработку программы более контролируемой. Другой мотив — это *повторное использование кода*, т.е. использова-

ние однажды написанных функций в качестве конструктивных блоков для создания новых программ. Многократное использование кода является основным определяющим фактором при переходе к объектно-ориентированному программированию. Имея хорошо продуманные имена и определения функций, можно создавать программы из стандартизованных модулей, не создавая специализированного программного кода. Эта методика известна как *абстракция*. Мы прибегаем к абстракции всякий раз, когда пишем программу, вызывающую функции стандартной библиотеки, например, **printf**, **scanf** и **pow**. Третьей причиной является правило избегать дублирования кода в программе. Оформление кода в виде функции позволяет выполнять его в разных частях программы посредством простого вызова функции.

Общее методическое замечание 5.3

Каждая функция должна ограничиваться выполнением одной, точно определенной задачи, а имя функции должно отражать смысл данной задачи. Это облегчает абстракцию и способствует многократному использованию программного кода.

Общее методическое замечание 5.4

Если вы не можете выбрать краткое имя, которое выражает назначение функции, возможно, что ваша функция пытается выполнить слишком много задач. Лучше разбить такую функцию на несколько меньших функций.

5.5. Определения функций

Каждая из приведенных ранее программ состояла из функции с именем **main**, которая для выполнения своей задачи вызывала функции стандартной библиотеки. Теперь мы рассмотрим, как программисты пишут свои собственные специализированные функции.

Рассмотрим программу, которая использует функцию **square** для вычисления квадрата целых чисел от 1 до 10 (рис. 5.3).

Функция **square** *активируется*, или *вызывается* в **main** внутри оператора **printf**:

```
printf("%d ", square(x));
```

Функция **square** получает копию значения **x** в *параметре* **y**. Затем **square** вычисляет значение произведения **y*y**. Результат передается назад функции **printf** в **main**, где была вызвана **square**, и **printf** выводит результат. Этот процесс повторяется десять раз в цикле **for**.

Определение функции **square** показывает, что **square** предполагает передачу в параметре **y** целого значения. Ключевое слово **int**, предшествующее имени функции, показывает, что **square** возвращает результат целого типа. Оператор **return** в **square** передает результат вычислений функции, которая вызвала **square**.

Строка программы

```
int square(int);
```

является *прототипом функции*. Ключевое слово **int** в скобках информирует компилятор о том, что **square** предполагает получить от вызывающей функции целое значение. Слово **int** слева от имени функции **square** информирует компилятор о том, что **square** возвращает вызывающей функции результат целого

типа. Компилятор использует прототип функции для проверки того, что вызовы **square** имеют корректный тип возвращаемых данных, корректное число аргументов, корректный тип аргументов и что аргументы следуют в правильном порядке. Прототипы функций детально рассматриваются в разделе 5.6.

```
/* Функция square, определенная программистом */
#include <stdio.h>

int sguare(int);    /* прототип функции */

main()
{
   int x;

   for (x = 1; x <= 10; x++)
      printf("%d   ", square(x));

   printf("\n");
   return 0;
}

/* Определение функции */
int square(int y)
{
   return y * y;
}
```

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

Рис. 5.3. Использование функции, определенной программистом

Хороший стиль программирования 5.3

Вставляйте пустую строку между определениями функций, чтобы отделить функции друг от друга и улучшить тем самым читаемость программы.

Определение функции имеет следующий формат:

```
тип_возвращаемого_значения имя_функции(список_параметров)
{
    объявления
    операторы
}
```

В качестве имени функции (*имя_функции*) может использоваться любой допустимый идентификатор. Типом результата, возвращаемого вызывающей функции, является *тип_возвращаемого_значения*. Если *тип_возвращаемого_значения* задан ключевым словом **void**, то это означает, что функция ничего не возвращает. Если *тип_возвращаемого_значения* не указан, то компилятор будет предполагать тип **int**.

Распространенная ошибка программирования 5.2

Если *тип_возвращаемого_значения* в определении функции опущен, то это вызовет синтаксическую ошибку в том случае, если прототип функции определяет, что возвращаемый тип не является целым (**int**).

Распространенная ошибка программирования 5.3

Если функция должна возвращать некоторое значение, а в функции опущен оператор возврата, то это может привести к непредвиденным ошибкам. Стандарт ANSI гласит, что в этом случае результат не определен.

Распространенная ошибка программирования 5.4

Возврат значения из функции, для которой возвращаемый тип был объявлен как **void**, вызывает синтаксическую ошибку.

Хороший стиль программирования 5.4

Хотя опущенный возвращаемый тип по умолчанию расценивается как **int**, всегда задавайте возвращаемый тип явным образом. Однако для функции **main** возвращаемый тип обычно опускается.

Список_параметров — это список объявлений параметров (отделенных запятыми), получаемых функцией при ее вызове. Если функция не получает значения, *список_параметров* обозначается ключевым словом **void**. Тип каждого параметра должен быть указан явно за исключением параметров типа **int**. Если тип параметра не указан, то по умолчанию принимается тип **int**.

Распространенная ошибка программирования 5.5

Запись **float x, y** вместо **float x, float y** при объявлении параметров функции, относящихся к одному типу. Такое объявление параметров, как **float x, y** фактически присвоило бы параметру **y** тип **int**, поскольку **int** принимается по умолчанию.

Распространенная ошибка программирования 5.6

Символ точки с запятой после правой круглой скобки, закрывающей список параметров в определении функции, является синтаксической ошибкой.

Распространенная ошибка программирования 5.7

Повторное определение параметра функции как локальной переменной внутри функции – синтаксическая ошибка.

Хороший стиль программирования 5.5

Включайте тип каждого параметра в список параметров функции даже в том случае, если данный параметр относится к типу **int**, принятому по умолчанию.

Хороший стиль программирования 5.6

Хотя это и не является ошибкой, не используйте одни и те же имена для передаваемых функции аргументов и соответствующих параметров в определении функции. Это поможет избежать неоднозначности.

Объявления и *операторы* внутри фигурных скобок образуют *тело функции*. Тело функции также называют *блоком*. Блок — это просто составной опе-

ратор, который включает в себя объявления. Переменные могут быть объявлены в любом блоке, а блоки могут быть вложены. *В любом случае функция не может быть определена внутри другой функции.*

Распространенная ошибка программирования 5.8

Определение функции внутри другой функции — синтаксическая ошибка.

Хороший стиль программирования 5.7

Выбор осмысленных имен функций и параметров делает программы более удобочитаемыми и помогает избежать чрезмерного использования комментариев.

Общее методическое замечание 5.5

Длина функции не должна превышать одной страницы. Еще лучше, чтобы функция не превышала по длине половины страницы. Небольшие функции способствуют повторному использованию программного кода.

Общее методическое замечание 5.6

Программы должны составляться в виде совокупности небольших функций. Это упрощает создание программ, их отладку, поддержку и модификацию.

Общее методическое замечание 5.7

Функция, требующая большого количества параметров, может выполнять слишком много задач. Рассмотрите возможность ее разделения на меньшие функции, которые выполняют выделенные задачи. Заголовок функции должен, по возможности, умещаться на одной строке.

Общее методическое замечание 5.8

Прототип функции, заголовок функции и вызов функции должны иметь взаимное соответствие по числу, типу и порядку следования аргументов и параметров, а также по типу возвращаемого значения.

Существует три способа возвращения управления в ту точку программы, в которой была вызвана функция. Если функция не возвращает результат, управление возвращается просто при достижении правой фигурной скобки, завершающей функцию, или при исполнении оператора

```
return;
```

Если функция возвращает результат, оператор

```
return выражение;
```

возвращает вызывающей функции значение *выражения*.

Наш второй пример демонстрирует определенную программистом функцию **maximum** для определения и возврата наибольшего из трех целых значений (рис. 5.4). С помощью функции **scanf** три целых числа вводятся. После этого числа передаются функции **maximum**, которая находит наибольшее целое число. Это значение возвращается в **main** оператором **return** в функции **maximum**. Возвращенное значение выводится на печать оператором **printf**.

```
/* Поиск максимального из трех целых значений */
#include <stdio.h>

int maximum(int, int, int);   /* прототип функции */

main()
{
   int a, b, c;

   printf("Enter three integers: ");
   scanf("%d%d%d", &a, &b, &c);
   printf("Maximum is: %d\n", maximum(a, b, c));

   return 0;
}

/* Определение функции maximum */
int maximum(int x, int y, int z)
{
   int max = x;

   if (y > max)
      max = y;

   if (z > max)
      max = z;

   return max;
}
```

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

Enter three integers: 85 22 17
Maximum is: 85

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

Рис. 5.4. Определенная программистом функция **maximum**

5.6. Прототипы функций

Одной из наиболее важных особенностей ANSI C являются *прототипы функций*. Они были позаимствованы комитетом ANSI C у разработчиков C++. Прототип функции сообщает компилятору тип данных, возвращаемых функцией, число параметров, получаемых функцией, тип и порядок следования параметров. Компилятор использует прототипы функций для проверки корректности обращений к функции. Предыдущие версии C не выполняли этот вид проверки, так что существовала возможность неправильного вызова функции, при котором компилятор не регистрировал ошибки. Такие обращения могли приводить к фатальным ошибкам при выполнении программы, или к ошибкам, которые, не вызывая краха программы, приводили тем не менее к труднообнаружимым логическим ошибкам. Прототипы функций в ANSI C исправляют такое положение дел.

Хороший стиль программирования 5.8

Включайте в программу прототипы для всех функций, чтобы воспользоваться преимуществами проверки типов С. Используйте директивы препроцессора **#include**, чтобы получить прототипы функций стандартной библиотеки из заголовочных файлов соответствующих библиотек. Также используйте **#include**, чтобы включить заголовочные файлы, содержащие ваши собственные прототипы функций или прототипы, используемые членами вашей рабочей группы.

Прототипом функции **maximum**, приведенной на рис. 5.4, является

```
int maximum(int, int, int);
```

Этот прототип объявляет, что **maximum** получает три параметра типа **int** и возвращает результат типа **int**. Обратите внимание, что прототип функции имеет тот же вид, что и первая строка определения функции **maximum**, за исключением того, что в него не включены имена параметров (**x**, **y** и **z**).

Хороший стиль программирования 5.9

Имена параметров иногда включаются в прототип функции с целью документирования. Компилятор игнорирует эти имена.

Распространенная ошибка программирования 5.9

Пропуск точки с запятой в конце прототипа функции вызывает синтаксическую ошибку.

Вызов функции, который не соответствует ее прототипу, вызывает синтаксическую ошибку. Ошибка генерируется также в том случае, если не согласованы прототип функции и ее функции. Например, если прототип функции записать в виде (рис. 5.4)

```
void maximum(int, int, int);
```

то компилятор зарегистрировал бы ошибку, поскольку возврат типа **void** в прототипе функции отличался бы от возврата типа **int** заголовка функции.

Другое важное следствие применения прототипов функций — *автоматическое приведение аргументов*, т.е. принудительное преобразование аргументов функции к соответствующему типу. Например, функция математической библиотеки **sqrt** может вызываться с целочисленным аргументом, и функция будет работать правильно, хотя прототип ее в **math.h** определяет аргумент как **double**. Оператор

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

правильно вычисляет **sqrt(4)** и выводит на печать значение **2.000**. Прототип функции заставляет компилятор преобразовать целое значение **4** в значение типа **double** — **4.0** — перед тем, как значение будет передано **sqrt**. В общем случае значения аргументов, которые не соответствуют в точности типам параметров в прототипе функции, перед вызовом преобразуются в соответствующий тип. Эти преобразования могут привести к неправильным результатам, если не учитывать *правила возведения типов* С. Правила возведения определяют, каким образом одни типы могут быть преобразованы в другие типы без потери данных. В нашем примере с **sqrt** тип **int** автоматически был преобразован в **double** без изменения значения. Однако преобразование типа **double** в тип **int** отбрасывает дробную часть значения **double**. Преобразование типа больших целых к типу малых целых (например, **long** в **short**) может также приводить к изменению значений.

Правила возведения автоматически применяются к выражениям, содержащим значения двух или более типов данных (называемым выражениями *смешанного типа*). Каждое значение в выражении смешанного типа автоматически возводится к наивысшему типу выражения[1] (фактически в выражении создается и используется временная версия каждого значения, а первоначальные значения остаются неизменными). На рис. 5.5 перечислены типы данных в порядке от наивысшего к низшему и приведены спецификации преобразования для операторов **printf** и **scanf**.

Тип данных	спецификация преобразования для printf	спецификация преобразования для scanf
long double	%Lf	%Lf
double	%f	%lf
float	%f	%f
unsigned long int	%lu	%lu
long int	%ld	%ld
unsigned int	%u	%u
int	%d	%d
short	%hd	%hd
char	%c	%c

Рис. 5.5. Иерархия возведения типов данных

Преобразование значений к более низким типам обычно приводит к неверному результату. Следовательно, значение может быть преобразовано в более низкий тип только посредством явного присваивания переменной более низкого типа или с помощью операции приведения. Значения аргументов функции преобразуются к типу параметров прототипа функции таким образом, как будто они непосредственно присваиваются переменным этого типа. Если наша функция **square**, которая имеет параметр целого типа (рис. 5.3), вызывается с аргументом типа **float**, то аргумент преобразуется в тип **int** (более низкий тип). В этом случае **square** может вернуть неверное значение. Например, **square(4.5)** дало бы значение **16**, а не **20.25**.

Распространенная ошибка программирования 5.10

Преобразование более высокого типа данных в иерархии возведения к более низкому типу может изменить значение данных.

Если прототип для данной функции не был включен в программу, компилятор формирует собственный прототип функции, используя ее первое вхождение в программу: или определение, или обращение к функции. По умолча-

1 Точнее говоря, процедура возведения типа применяется *последовательно* к каждой паре операндов текущей операции *по ходу оценки выражения*. Например, оценка выражения 3/2*3.14 даст в результате 3.14, так как для первой операции (/) никакого возведения не производится (оба операнда целые) и результатом ее будет целое 1. Только на следующем шаге этот промежуточный результат будет возведен до 1.0, поскольку второй операнд умножения — число с плавающей точкой. — *Прим. ред.*

нию компилятор предполагает, что функция возвращает тип **int**, ничего не предполагая относительно типа аргументов. Следовательно, если аргументы, переданные функции, некорректны, то компилятор не обнаружит ошибку.

Распространенная ошибка программирования 5.11

Пропуск прототипа функции вызывает синтаксическую ошибку, если функция возвращает тип, отличный от **int**, а определение функции появляется в программе после вызова функции. В других случаях пропуск прототипа функции может вызвать ошибку во время выполнения программы или привести к непредвиденному результату.

Общее методическое замечание 5.9

Прототип функции, размещенный вне любого определения функции, применяется ко всем вызовам, появляющимся в файле после прототипа функции. Прототип функции, помещенный в функцию, применяется только к тем вызовам, которые делаются из этой функции.

5.7. Заголовочные файлы

Каждая стандартная библиотека имеет свой *заголовочный файл*, содержащий прототипы для всех функций данной библиотеки, а также определения различных типов данных и констант, необходимых этим функциям. На рис. 5.6 приведен алфавитный список заголовочных файлов стандартной библиотеки, которые могут быть включены в программу. Термин «макрос», который несколько раз используется в рис. 5.6, подробно обсуждается в главе 13, «Препроцессор».

Заголовочный файл стандартной библиотеки	Содержимое заголовочного файла
`<assert.h>`	Содержит диагностические макросы и информацию, которая помогает при отладке программы.
`<ctype.h>`	Содержит прототипы для функций, которые проверяют определенные характеристики символов и прототипы для функций, которые могут использоваться для преобразования символов нижнего регистра в символы верхнего регистра и наоборот.
`<errno.h>`	Определяет макросы, которые полезны для сообщений о типе ошибки.
`<float.h>`	Содержит пределы значений для чисел с плавающей точкой в системе.
`<limits.h>`	Содержит пределы значений для целых чисел в системе.
`<locale.h>`	Содержит прототипы функций и другую информацию, которая позволяет изменять работу программы согласно правилам текущего локала, в котором она выполняется. Понятие локала дает компьютерной системе возможность обрабатывать соглашения, принятые в различных странах по всему миру для выражения таких данных, как дата, время, денежные единицы и способ записи больших чисел.
`<math.h>`	Содержит прототипы для функций математической библиотеки.
`<setjmy.h>`	Содержит прототипы для функций, которые позволяют обойти обычный вызов функции и последовательность возврата.

Заголовочный файл стандартной библиотеки	Содержимое заголовочного файла
<signal.h>	Содержит прототипы функций и макросы для обработки различных условий, которые могут возникать во время выполнения программы.
<stdarg.h>	Определяет макросы для обработки списка параметров функции, для которой неизвестно число параметров и их тип.
<stddef.h>	Содержит общие определения типов, используемых С для выполнения некоторых вычислений.
<stdio.h>	Содержит прототипы для функций ввода/вывода стандартной библиотеки и информацию, используемую ими.
<stdlib.h>	Содержит прототипы функций для преобразования чисел в текст и текста в числа, прототипы функций размещения памяти, генерации случайных чисел и других сервисных функций.
<string.h>	Содержит прототипы для функций обработки строки.
<time.h>	Содержит прототипы функций и типы для функций управления временем и датой.

Рис. 5.6. Заголовочные файлы стандартной библиотеки

Программист может создать специализированный заголовочный файл. Определенные программистом заголовочные файлы также должны заканчиваться **.h**. Определенный программистом файл может быть включен директивой препроцессора **#include**. Например, заголовочный файл **square**.h может быть включен в нашу программу с помощью директивы

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

размещенной в верхней части программы. В разделе 13.2 представлена дополнительная информация по включению в программу заголовочных файлов.

5.8. Вызов функций: вызов по значению и по ссылке

Во многих языках программирования существует два способа вызова функций — *вызов по значению* и *вызов по ссылке*. Когда аргумент используется в вызове по значению, то вызываемой функции передается *копия* значения аргумента. Изменения, происходящие с копией, не отражаются на значении исходной переменной в вызывающей функции. Когда аргумент функции передается по ссылке, вызывающая функция фактически позволяет вызываемой функции изменять значение исходной переменной.

Передача аргумента по значению должна использоваться, когда вызываемой функции не нужно менять значение исходной переменной в вызывающей функции. Это предотвращает случайные *побочные эффекты*, которые мешают разработке правильных и надежных систем программного обеспечения. Передача аргумента по ссылке должна применяться только с такими вызываемыми функциями, которым необходимо менять первоначальную переменную и которым можно «доверять».

В С все вызовы передают аргументы по значению. Как мы увидим в главе 7, имеется возможность *имитировать* передачу аргумента по ссылке, используя операции взятия адреса и косвенные операции. В главе 6 мы увидим,

что массивы автоматически передаются посредством имитации передачи аргумента по ссылке. Мы должны подождать до главы 7, чтобы полностью разобраться в этом довольно сложном вопросе. Пока мы ограничимся вызовами с передачей аргументов по значению.

5.9. Генерация случайных чисел

Теперь мы сделаем небольшой перерыв и немного развлечемся с интересным и популярным приложением программирования, а именно с моделированием и играми. В этом и следующем разделах мы разработаем хорошо структурированную игровую программу, которая включает в себя много функций. Программа демонстрирует большую часть изученных нами управляющих структур.

В азартной атмосфере казино есть что-то, что определяет поведение любого оказавшегося там человека: от игрока, делающего крупные ставки за дорогим столом для игры в кости, до игрока за игральным автоматом. Это — *элемент случайности*, возможность того, что случай превратит просто полный карман в гору денег. Элемент случайности может быть введен и в компьютерные приложения, если использовать функцию **rand** из стандартной библиотеки С.

Рассмотрим следующий оператор:

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

Функция **rand** генерирует целое значение в диапазоне от 0 до **RAND_MAX** (символическая константа, определенная в заголовочном файле **<stdlib.h>**). Стандарт ANSI объявляет, что значение **RAND_MAX** должно быть равно по меньшей мере 32767, т.е. максимальному значению 2-байтового целого (или 16-битового, что то же самое). Программы этого раздела были проверены на системе С с максимальным значением 32767 для **RAND_MAX**. Если **rand** действительно генерирует целые числа случайным образом, то каждое число между **0** и **RAND_MAX** имеет равные шансы (или *вероятность*) быть выбранным при каждом вызове функции **rand**.

Диапазон значений, непосредственно генерируемых **rand**, часто отличается от того, который необходим в данном приложении. Например, программе, которая имитирует подбрасывание монеты, требуется только два значения: 0 для «орла» и 1 для «решки». Программе, которая моделирует выбрасывание игральной кости с шестью сторонами, потребовались бы случайные целые числа в диапазоне от 1 до 6.

Чтобы продемонстрировать функцию **rand**, давайте разработаем программу, которая имитирует 20 бросков шестигранной игральной кости и печатает значение, выпавшее при каждом броске. Прототип функции для **rand** может быть найден в **<stdlib.h>**. Мы применим операцию взятия по модулю (%) в сочетании с **rand**:

```
rand() % 6
```

чтобы генерировать целые в диапазоне от 0 до 5. Это называется *масштабированием*. Число 6 называется коэффициентом масштабирования. После этого мы *сдвигаем* диапазон генерируемых чисел, добавляя 1 к нашему предыдущему результату. Рис. 5.7 подтверждает, что результаты находятся в диапазоне от 1 до 6.

Чтобы показать, что эти числа появляются приблизительно с равной вероятностью, давайте смоделируем 6000 бросков игральной кости, используя программу, приведенную на рис. 5.8. Каждое целое число от 1 до 6 должно «выпасть» приблизительно 1000 раз.

```
/* Генерация целых 1 + rand() % 6 в масштабе и со сдвигом */
#include <stdio.h>
#include <stdlib.h>

main()
{
   int i;

   for (i = 1; i <= 20; i++) {
      printf("%10d", 1 + (rand() % 6));

      if (i % 5 == 0)
         printf("\n");
   }

   return 0;
}
```

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

Рис. 5.7. Масштабированные и сдвинутые целые, генерируемые выражением **1 + rand() % 6**

```
/* Бросание шестигранной игральной кости 6000 раз */
#include <stdio.h>
#include <stdlib.h>

main()
{
   int face, roll, frequency1 = 0, frequency2 = 0,
      frequency3 = 0, frequency4 = 0,
      frequency5 = 0, frequency6 = 0;

   for (roll = 1; roll <= 6000; roll++) {
      face = 1 + rand() % 6;

      switch (face) {
         case 1:
            ++frequency1;
            break;
         case 2:
            ++frequency2;
            break;
         case 3:
            ++frequency3;
            break;
         case 4:
            ++frequency4;
            break;
         case 5:
            ++frequency5;
            break;
```

Рис. 5.8. Бросание шестигранной игральной кости 6000 раз (часть 1 из 2)

```
         case 6:
            ++frequency6;
            break;
      }
   }

   printf("%s%13s\n", "Face", "Frequency");
   printf("  1%13d\n", frequency1);
   printf("  2%13d\n", frequency2);
   printf("  3%13d\n", frequency3);
   printf("  4%13d\n", frequency4);
   printf("  5%13d\n", frequency5);
   printf("  6%13d\n", frequency6);
   return 0;
}
```

```
Face      Frequency
   1            987
   2            984
   3           1029
   4            974
   5           1004
   6           1022
```

Рис. 5.8. Бросание шестигранной игральной кости 6000 раз (часть 2 из 2)

Как показывает результат, выведенный программой, масштабирование и сдвиг в сочетании с функцией **rand** позволяет реалистично моделировать вращение шестигранной игральной кости. Обратите внимание, что в структуре **switch** *нет варианта* **default**. *Также обратите внимание на использование спе*цификации преобразования **%s** для печати символьных строк «**Face**» и «**Frequency**» в качестве заголовков столбцов. После того как мы исследуем массивы в главе 6, мы покажем, как заменить всю структуру **switch** на оператор, умещающийся в одной строке.

Повторное выполнение программы рис. 5.7 дает результат

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

Обратите внимание, что была напечатана та же самая последовательность значений. Как же это может быть случайными числами? Как ни странно, эта повторяемость является важной характеристикой функции **rand**. При отладке программы эта повторяемость является существенным фактором для доказательства того, что исправления, внесенные в программу, работают правильно.

Функция **rand** генерирует на самом деле *псевдослучайные числа*. При вызове **rand** генерируется последовательность чисел, которые выглядят случайными. Однако эта последовательность повторяется при каждом новом выполнении программы. Когда программа будет тщательно отлажена, можно модифицировать ее для генерации различных последовательностей случайных чисел для каждого выполнения. Это называется *рандомизацией* и обеспечивается функцией стандартной библиотеки **srand**. Функция **srand** получает в качестве аргумента целое без знака (тип **unsigned**), называемое *семенем*, которое

позволяет получать от **rand** различные последовательности случайных чисел *при каждом исполнении программы.*

Применение функции **srand** показано на рис. 5.9. В программе мы используем тип данных **unsigned**, что является сокращением записи **unsigned int**. Тип **int** хранится по меньшей мере в двух байтах памяти и может принимать положительные и отрицательные значения. Переменная типа **unsigned** также хранится по меньшей мере в двух байтах памяти. Двухбайтовое целое типа **unsigned int** может принимать только положительные значения в диапазоне от 0 до 65535. Четырехбайтовое целое типа **unsigned int** может принимать положительные значения в диапазоне от 0 до 4294967295. Функция **srand** получает в качестве аргумента значение типа **unsigned**. Для чтения данных типа **unsigned** функцией **scanf** необходима спецификация преобразования **%u**. Прототип функции для **srand** находится в **<stdlib.h>**.

Давайте выполним программу несколько раз и посмотрим на результаты. Обратите внимание, что при каждом выполнении программы получаются *различные* последовательности случайных чисел, что обеспечивается вводом различных значений для семени.

```
/* Рандомизация программы бросания игральной кости */
#include <stdlib.h>
#include <stdio.h>

main()
{
   int I;
   unsigned seed;

   printf("Enter seed: ");
   scanf("%u", &seed);
   srand(seed);

   for (i = 1; i <= 10; i++) {
      printf("%10d", 1 + (rand() % 6));

      if (i % 5 == 0)
         printf("\n");
   }

   return 0;
}
```

```
Enter seed:67
         1         6         5         1         4
         5         6         3         1         2

Enter seed: 432
         4         2         5         4         3
         2         5         1         4         4

Enter seed: 67
         1         6         5         1         4
         5         6         3         1         2
```

Рис. 5.9. Рандомизация программы бросания игральной кости (окончание)

Если мы хотим проводить рандомизацию без ввода семени, мы можем написать оператор вроде

```
srand(time(NULL));
```

Компьютер здесь считывает показания внутренних часов, чтобы автоматически получить не определенное заранее значение. Функция **time** возвращает текущее время дня в секундах[1]. Это значение преобразуется в целое без знака и используется как семя для генератора случайных чисел. Функция **time** получает в качестве аргумента **NULL** (функция **time** может формировать строку, представляющую время дня[2]; аргумент **NULL** запрещает этот вариант в данном вызове **time**). Прототип функции **time** находится в **<time.h>**.

Значения, непосредственно генерируемые **rand**, всегда находятся в диапазоне

```
0 <= rand() <= RAND_MAX
```

Предварительно мы показали, как написать единственный оператор С, имитирующий вращение шестигранной игральной кости:

```
face = 1 + rand() % 6;
```

Этот оператор всегда присваивает целое значение (случайным образом) переменной **face** в диапазоне от 1 до 6. Обратите внимание, что ширина этого диапазона (т.е. число последовательных целых чисел в нем) равна 6 и начальным числом этого диапазона является 1. Что касается предшествующего утверждения, мы видим, что ширина диапазона определяется числом, используемым для масштабирования функции **rand** операцией деления по модулю (т.е. 6), а начальное число диапазона равно числу, которое прибавляется к **rand % 6** (т.е. **1**). Мы можем обобщить этот результат следующим образом:

```
n = a + rand() % b;
```

Здесь **a** — *значение сдвига* (которое равно первому числу диапазона последовательных целых чисел), и **b** — коэффициент масштабирования (который равен ширине диапазона последовательных целых чисел). В упражнениях в конце раздела мы увидим, что можно выбирать случайные целые числа из наборов значений, не являющихся последовательными.

Распространенная ошибка программирования 5.12

Использование **srand** вместо **rand** для генерации случайных чисел.

5.10. Пример: стохастическая игра

Одной из наиболее популярных стохастических игр является «крепс», в который играют в казино и в темных закоулках по всему миру. Правила игры просты:

Игрок выбрасывает две кости. Каждая кость имеет шесть граней. Эти грани содержат 1, 2, 3, 4, 5 и 6 точек. После того как кости остановятся, вычисляется сумма точек на двух гранях, повернутых вверх. Если выпавшая сумма при первом броске оказалась равной 7 или 11, то победил игрок. Если

1 Хоть в данном примере это и не важно, заметим, что функция time возвращает не текущее время дня, а время (в секундах), прошедшее с момента 00:00:00 GMT 1-го января 1970 г. — *Прим. ред.*

2 Функция time этого не может. Ее аргумент (если он не NULL) — это просто указатель на переменную, в которую записывается то же самое значение, что функция возвращает. — *Прим. ред.*

сумма при первом броске составила 2, 3 или 12, то игрок проигрывает (выигрывает казино). Если сумма первого броска равна 4, 5, 6, 8, 9 или 10, то эта сумма становится очками игрока. Чтобы выиграть, вы должны продолжить бросать кости до тех пор, пока не наберете свои очки еще раз. Игрок проигрывает, если при очередном броске выпадает 7.

Программа, приведенная на рис. 5.10, моделирует данную игру. На рис. 5.11 показано несколько примеров выполнения программы.

```
/* Игра в крепс */
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int rollDice(void);

main()
{
   int gameStatus, sum, myPoint;

   srand(time(NULL));
   sum = rollDice();                  /* первый выброс костей */

   switch(sum) {
      case 7: case 11:                /* выигрыш при первом броске */
         gameStatus = 1;
         break;
      case 2: case 3: case 12:        /* проигрыш при первом броске */
         gameStatus = 2;
         break;
      default:                        /* запомнить очки */
         gameStatus = 0;
         myPoint = sum;
         printf("Point is %d\n", myPoint);
         break;
   }
   while (gameStatus == 0) {          /* продолжать бросать */
      sum = rollDice();

      if (sum == myPoint)/* выигрыш повторным выбросом очков */
         gameStatus = 1;
      else
         if (sum == 7)                /* проигрыш при выбросе 7 */
         gameStatus = 2;
   }
   if (gameStatus == 1)
      printf("Player wins\n");
   else
      printf("Player loses\n");

   return 0;
}
int rollDice(void)
{
   int die1, die2, workSum;
```

Рис. 5.10. Программа, моделирующая игру в крепс (часть 1 из 2)

```
    die1 = 1 + (rand() % 6);
    die2 = 1 + (rand() % 6);
    workSum = die1 + die2;
    printf("Player rolled %d + %d = %d\n", die1, die2, workSum);
    return workSum;
}
```

Рис. 5.10. Программа, моделирующая игру в крепс (часть 2 из 2)

```
Player rolled 6 + 5 = 11
Player wins
Player rolled 6 + 6 = 12
Player loses

Player rolled 4 + 6 = 10
Point is 10

Player rolled 2 + 4 = 6
Player rolled 6 + 5 = 11
Player rolled 3 + 3 = 6
Player rolled 6 + 4 = 10
Player wins

Player rolled 1 + 3 = 4
Point is 4
Player rolled 1 + 4 = 5
Player rolled 5 + 4 = 9
Player rolled 4 + 6 = 10
Player rolled 6 + 3 = 9
Player rolled 1 + 2 = 3
Player rolled 5 + 2 = 7
Player loses
```

Рис. 5.11. Пример выполнения программы игры в кости

Обратите внимание, что и в первом броске и во всех последующих игрок выбрасывает две кости. Мы определяем функцию **rollDice** таким образом, что она выполняет броски игровых костей, вычисляет и печатает сумму. Функция **rollDice** определена один раз, но вызывается в двух местах программы. Интересно, что **rollDice** не получает никаких аргументов, поскольку в списке параметров стоит **void**. Функция **rollDice** возвращает сумму, выпавшую на двух игровых костях, так что в заголовке функции в качестве возвращаемого типа указан тип **int**.

Начинается игра. Игрок может выиграть либо проиграть уже при первом броске, или может выиграть либо проиграть при любом последующем. Для отслеживания ситуации используется переменная **gameStatus**.

Когда программа выиграла либо на первом броске, либо на любом из последующих, переменная **gameStatus** устанавливается равной 1. Когда программа проиграла на первом броске либо на одном из последующих, **gameStatus** устанавливается равной 2. В других случаях **gameStatus** равна нулю и игра должна продолжаться.

Если после первого броска игра заканчивается, программа обходит структуру **while**, поскольку переменная **gameStatus** не равна нулю. Программа переходит к структуре **if/else**, которая выводит на печать строку **«Player wins» (Игрок выиграл)**, если **gameStatus** равна **1**, и **«Player loses» (Игрок проиграл)**, если **gameStatus** равна **2**.

Если после первого броска игра не заканчивается, то значение **sum** сохраняется в переменной **myPoint**. Выполнение программы продолжается с участием структуры **while**, поскольку **gameStatus** равна **0**. Каждый раз при прохождении структуры **while** вызывается функция **rollDice**, которая генерирует новую сумму. Если значение **sum** совпадает со значением **myPoint**, **gameStatus** устанавливается на **1**, чтобы показать, что игрок выиграл. После этого следует выход из структуры **while**, поскольку ее условие перестает выполняться, структура **if/else** выводит на печать строку **«Player wins»** и выполнение программы прекращается. Если значение **sum** равно **7**, **gameStatus** устанавливается на **2**, чтобы показать, что игрок проиграл. Далее следует выход из структуры **while**, оператор **if/else** печатает строку **«Player loses»** и выполнение программы прекращается.

Обратите внимание на интересную управляющую структуру программы. Мы использовали две функции — **main** и **rollDice**, структуры **switch**, **while**, **if/else** и вложенную структуру **if**. В упражнениях мы исследуем различные интересные характеристики игры в крепс.

5.11. Классы памяти

В главах 2–4 для имен переменных мы использовали идентификаторы. Атрибуты переменных включают в себя имя, тип и значение. В этой главе мы также используем идентификаторы в качестве имен функций, определяемых пользователем. На самом деле каждый идентификатор в программе имеет и другие атрибуты, включая *класс памяти*, *период хранения*, *область действия* и *тип компоновки*.

Язык C поддерживает четыре класса памяти, обозначаемые *спецификаторами класса памяти*: ***auto***, ***register***, ***extern*** и ***static***. Класс памяти идентификатора помогает определить его период хранения, область действия и тип компоновки. *Период хранения* идентификатора — это время, в течение которого данный идентификатор существует в памяти. Некоторые идентификаторы существуют короткое время, некоторые неоднократно создаются и разрушаются, другие существуют в течение всего времени выполнения программы. *Область действия* идентификатора характеризует возможность обращения к нему из различных частей программы. Некоторые идентификаторы доступны во всей программе, другие — только в отдельных ее частях. *Тип компоновки* идентификатора определяется для программ, состоящих из нескольких исходных файлов, объединяемых на этапе компоновки (предмет, который мы исследуем в главе 14). Эта характеристика показывает, известен ли идентификатор только в текущем исходном файле или в любом исходном файле с соответствующими объявлениями. В этом разделе рассматриваются четыре класса памяти и период хранения. В разделе 5.12 обсуждается область действия идентификаторов. В главе 14 обсуждается тип компоновки идентификатора и программирование с несколькими исходными файлами.

Четыре спецификатора класса памяти могут быть разбиты на два типа по периоду хранения: *автоматический период хранения* и *статический период хранения*. Для объявления переменных с автоматическим периодом хранения служат ключевые слова **auto** и **register**. Переменные с автоматическим хранением создаются, когда управление получает программный блок, в котором они объявлены. Переменные этого типа существуют, пока блок активен, и уничтожаются, когда происходит выход из блока.

Автоматический период хранения могут иметь только переменные. Локальные переменные функции (объявленные в списке параметров или в теле функции) обычно имеют автоматический период хранения. Ключевое слово **auto** объявляет переменные с автоматическим хранением явным образом. Например, следующее объявление показывает, что переменные **x** и **y** типа **float** являются автоматическими локальными переменными и существуют только в теле функции, в которой находится данное объявление:

```
auto float x, y;
```

Локальные переменные имеют автоматический период хранения по умолчанию, так что ключевое слово **auto** используется редко. Во всем последующем тексте мы будем называть переменные с автоматическим периодом хранения просто автоматическими переменными.

Совет по повышению эффективности 5.1

Автоматическое хранение способствует экономии памяти, поскольку автоматические переменные существуют только тогда, когда они необходимы. Они создаются при запуске функции, в которой они объявлены, и уничтожаются, когда происходит выход из функции.

Общее методическое замечание 5.10

Автоматическое хранение является примером реализации *принципа минимальных привилегий*. Почему переменные должны храниться в памяти и быть доступными, когда в данный момент в них нет необходимости?

Данные в программе на машинном языке для выполнения вычислений и другой обработки обычно загружаются в регистры.

Совет по повышению эффективности 5.2

Перед объявлением автоматической переменной может быть помещен спецификатор класса памяти **register**, чтобы рекомендовать компилятору разместить ее в одном из быстродействующих аппаратных регистров компьютера. Если интенсивно используемые переменные типа счетчиков или сумм будут реализованы в аппаратных регистрах, то можно исключить непроизводительные затраты на неоднократную загрузку переменных из памяти в регистры и обратно.

Компилятор может проигнорировать объявление **register**. Например, в распоряжении компилятора может не оказаться достаточного числа регистров. Следующее объявление предлагает, чтобы целая переменная **counter** была размещена в одном из регистров компьютера и инициализирована значением **1**:

```
register int counter = 1;
```

Ключевое слово **register** может использоваться только с переменными, имеющими автоматический период хранения.

Совет по повышению эффективности 5.3

Объявления **register** часто бывают не нужны. Современные оптимизирующие компиляторы способны распознать часто используемые переменные и могут размещать их в регистрах самостоятельно, не требуя от программиста объявления **register**.

Ключевые слова **extern** и **static** используются для объявления идентификаторов переменных и функций со статическим периодом хранения. Идентификаторы статического периода хранения существуют с того момента, как программа начинает выполняться. Для переменных память распределяется и инициализируется один раз, когда программа запускается. Для функций имя также начинает существовать с момента начала выполнения программы. Однако даже при том, что переменные и имена функций существуют с момента начала выполнения программы, это не означает, что эти идентификаторы доступны во всей программе. Период хранения и область действия (область, где имя доступно) являются разными вещами, как мы и увидим в разделе 5.12.

Существует два типа идентификаторов со статическим периодом хранения: внешние идентификаторы (вроде глобальных переменных и имен функций) и локальные переменные, объявленные со спецификатором класса памяти **static**. Глобальные переменные и имена функций имеют по умолчанию класс памяти **extern**. Глобальные переменные создаются при помещении их объявлений вне любого определения функции, и они сохраняют свои значения в течение всего времени выполнения программы. Обращение к глобальным переменным и функциям возможно из любой функции, которая следует после их объявления или определения в файле. Это является одной из причин использования прототипов функций. Когда мы включаем **stdio.h** в программу, которая вызывает **printf**, прототип функции помещается в начало нашего файла, делая имя **printf** известным для остальной части файла.

Общее методическое замечание 5.11

Объявление переменной как глобальной, а не локальной, может приводить к случайным побочным эффектам, когда функция, которой не нужен доступ к этой переменной, случайно или преднамеренно ее изменяет. В целом нужно избегать использования глобальных переменных, за исключением некоторых ситуаций с уникальными требованиями к производительности (рассматривается в главе 14).

Хороший стиль программирования 5.10

Переменные, использующиеся только в определенной функции, должны быть объявлены локальными переменными этой функции, а не внешними переменными.

Локальные переменные, объявленные с ключевым словом **static**, остаются известными только той функции, в которой они определены, но в отличие от автоматических статические локальные переменные сохраняют свое значение и после выхода из функции. При следующем вызове статическая локальная переменная будет содержать то значение, которое она имела при последнем выходе из функции. Следующий оператор объявляет, что локальная переменная **static** будет статической, и инициализирует ее значением **1**.

```
static int count = 1;
```

Все числовые переменные со статическим хранением инициализируются нулем, если они явно не инициализированы программистом. (Переменные-указатели, обсуждаемые в главе 7, инициализируются значением **NULL**.)

Распространенная ошибка программирования 5.13

Использование нескольких спецификаторов класса памяти для идентификатора. К данному идентификатору может применяться только один спецификатор класса памяти.

Ключевые слова **extern** и **static** имеют специальное значение, когда применяются к внешним идентификаторам явным образом. В главе 14 мы увидим программы, явно использующие **extern** и **static**.

5.12. Правила области действия

Областью действия идентификатора является та часть программы, в которой возможно обращение к нему. Например, когда мы объявляем в некотором блоке локальную переменную, к ней можно обратиться только из этого блока или из блоков, вложенных в данный блок. Область действия идентификатора делится на четыре вида: *область действия функции*, *область действия файла*, *область действия блока* и *область действия прототипа функции*.

Метки (идентификаторы, сопровождающиеся двоеточием, вроде **start:**) являются единственными идентификаторами, принадлежащими *области действия функции*. Метки могут использоваться в произвольном месте функции, в которой они появляются, но на них нельзя сослаться вне тела функции. Метки используются в структурах **switch** (в качестве меток **case**) и в операторах **goto** (см. главу 14). Метки относятся к подробностям реализации, которые функции скрывают от друг друга. Эта скрытность, более формально называемая *сокрытием информации*, является одним из наиболее фундаментальных принципов хорошего стиля программирования.

Идентификатор, объявленный вне любой функции, имеет *область действия файла*. Такой идентификатор «известен» всем функциям начиная с того места, где он объявлен, и до конца файла. Глобальные переменные, определения функций и прототипы функций, помещенные вне функции, — все они имеют область действия файла.

Идентификаторы, объявленные внутри блока, имеют *область действия блока*. Область действия блока заканчивается завершающей правой фигурной скобкой (}) блока. Локальные переменные, объявленные в начале функции, имеют область действия блока, так же как и параметры функции, которые рассматриваются как ее локальные переменные. Любой блок может содержать объявления переменных. Когда блоки вложены, а идентификатор во внешнем блоке имеет то же самое имя, что и идентификатор во внутреннем блоке, идентификатор во внешнем блоке «скрывается», пока внутренний блок не завершит работу. Это означает, что пока выполняется внутренний блок, он видит значение собственного локального идентификатора, а не значение идентификатора с тем же именем, находящегося в объемлющем блоке. Локальные переменные, объявленные как static, имеют область действия блока несмотря на то, что существуют с момента начала выполнения программы. Таким образом, период хранения не влияет на область действия идентификатора.

Единственными идентификаторами с *областью действия прототипа функции* являются идентификаторы, которые используются в списке параметров прототипа функции. Как отмечалось выше, прототипы функций не требуют, чтобы в списке параметров стояли имена идентификаторов; требуется только их тип. Если в списке параметров прототипа используется имя, то компилятор его игнорирует. Идентификаторы, указанные в прототипе функции, могут неоднократно встречаться в других местах программы, и здесь не возникает никакой неоднозначности.

Распространенная ошибка программирования 5.14

Случайное объявление во внутреннем блоке имени идентификатора, которое уже используется во внешнем блоке, в то время как программист на самом деле хотел, чтобы идентификатор внешнего блока был доступен для внутреннего блока.

Хороший стиль программирования 5.11

Не объявляйте имен переменных, которые скрывают имена во внешних областях действия. Можно просто избегать любого дублирования идентификаторов в программе.

Программа, приведенная на рис. 5.12, иллюстрирует правила применения глобальных, автоматических локальных и статических локальных переменных. Глобальная переменная **x** объявляется и инициализируется значением 1. Эта глобальная переменная скрыта в любом блоке (или функции), в котором объявлена переменная с именем **x**. В **main** объявляется и инициализируется значением 5 локальная переменная **x**. Эта переменная сразу выводится на печать, чтобы показать, что глобальная переменная **x** скрыта от **main**. После этого в **main** определяется новый блок с другой локальной переменной **x**, инициализированной значением 7. Эта переменная выводится на печать, чтобы показать, что переменная **x** нового блока скрывает переменную внешнего блока **main**. Переменная **x** со значением 7 автоматически разрушается, когда блок прекращает работу, и локальная переменная x внешнего блока **main** печатается еще раз, чтобы показать, что она уже не скрыта. В программе определены три функции таким образом, что каждая из них не получает никаких аргументов и ничего не возвращает. Функция **a** определяет автоматическую локальную переменную **x** и инициализирует ее значением 25. При вызове функции **a** эта переменная выводится на печать, увеличивается и печатается еще раз перед выходом из функции. Каждый раз при вызове этой функции автоматическая локальная переменная **x** заново инициализируется значением 25. Функция **b** объявляет статическую переменную **x** и инициализирует ее значением 50. Локальные переменные, объявленные как **static**, сохраняют свои значения даже вне обычной области действия локальной переменной. При вызове функции **b** печатается значение **x**, затем **x** увеличивается и печатается еще раз перед выходом из функции. При следующем обращении к этой функции статическая локальная переменная **x** будет содержать значение 51. Функция c не объявляет никаких переменных. Следовательно, когда она обращается к переменной **x**, используется глобальная переменная **x**. При следующем вызове функции c печатается значение глобальной переменной, она умножается на 10 и печатается еще раз перед выходом из функции. При последующем вызове функции c глобальная переменная все еще имеет модифицированное значение — 10. В заключение программа печатает значение локальной переменной **x** в **main** еще раз, чтобы показать, что ни один из вызовов функций не изменил значение **x**, поскольку функции ссылались на переменные в других областях действия.

5.13. Рекурсия

Программы, которые мы обсуждали, имели в основном структуру функций, которые вызывают другие функции, подчиняясь строгой иерархии. Для некоторых типов задач полезно иметь функции, которые вызывают сами себя. *Рекурсивная функция* — это функция, которая вызывает саму себя или непосредственно, или косвенно через другую функцию. Рекурсия — сложный пред-

мет, изучаемый лишь в некоторых курсах по компьютерным дисциплинам. В этом и в следующих разделах приведены простые примеры рекурсии. Эта книга уделяет большое внимание рекурсии, которая так или иначе применяется начиная с главы 5 и до главы 12. Рис. 5.17 в конце раздела 5.15 содержит сводку по всем примерам и упражнениям на рекурсию, приведенным в книге.

```
/* Пример по областям действия */
#include <stdio.h>

void a(void); /* прототип функции */
void b(void); /* прототип функции */
void c(void); /* прототип функции */

int x = 1;/* глобальная переменная */

main()
{
   int x = 5; /* локальная переменная для main */

   printf ("local x in outer scope of main is %d\n", x);

   {           /* начало новой области действия */
      int x = 7;

      printf("local x in inner scope of main is  %d\n", x);
   }           /* конец новой области действия */

   printf("local x in outer scope of main is  %d\n", x);

   a();        /* a содержит автоматическую локальную x */
   b();        /* b содержит статическую локальную x */
   c();        /* c использует глобальную x */
   a();        /* a заново инициализирует автоматическую
                  локальную x */
   b();        /* статическая локальная x сохраняет предыдущее
                  значение */
   c();        /* глобальная x также сохраняет свое значение */

   printf("local x in main is %d\n", x);
   return 0;
}

void a(void)
{
   int x = 25; /* инициализируется при каждом вызове a */

      printf("\nlocal x in a is %d after entering a\n", x);
      ++x;
      printf("local x in a is %d before exiting a\n", x);
}

void b(void)
{
   static int x = 50;     /* инициализация как static */
                          /* только при первом вызове b */
   printf("\nlocal static x is %d on entering b\n", x);
```

Рис 5.12. Программа, иллюстрирующая области действия (часть 1 из 2)

```
   ++x;
   printf("local static x is %d on exiting b\n", x);
}

void c(void)
{
   printf("\nglobal x is %d on entering c\n", x);
   x *= 10;
   printf("global x is %d on exiting c\n", x);
}
```

```
local x in outer scope of main is 5
local x in inner scope of main is 7
local x in outer scope of main is 5

local x in a is 25 after entering a
local x in a is 26 before exiting a

local static x is 50 on entering b
local static x is 51 on exiting b

global x is 1 on entering c
global x is 10 on exiting c

local x in a is 25 after entering a
local x in a is 26 before exiting a

local static x is 51 on entering b
local static x is 52 on exiting b

global x is 10 on entering c
global x is 100 on exiting c
local x in main is 5
```

Рис. 5.12. Программа, иллюстрирующая области действия (часть 2 из 2)

Сначала мы рассмотрим рекурсию обобщенно, а затем исследуем несколько программ, содержащих рекурсивные функции. Рекурсивные подходы к решению задач имеют ряд общих элементов. Для решения задачи вызывается рекурсивная функция. Фактически функция знает решение только простейшего случая (случаев), или так называемого *основного случая.* Если функция вызывается для решения основного случая, то она просто возвращает результат. Если функция вызывается для решения более сложной проблемы, функция делит задачу на две обобщенных части: часть, для которой функция имеет способ решения, и часть, для которой функция решения не имеет. Чтобы сделать рекурсию возможной, последняя часть должна быть похожа на первоначальную задачу, но она должна быть более простым, или редуцированным ее вариантом. Поскольку эта новая задача похожа на первоначальную, функция запускает (вызывает) свою новую копию, чтобы продолжить решение меньшей задачи. Этот процесс называется *рекурсивным вызовом* или *шагом рекурсии.* Шаг рекурсии также включает ключевое слово **return**, поскольку ее результат будет объединен с предыдущей частью задачи, для которой функция знала решение, чтобы сформировать окончательный результат, который будет передан первоначальной вызывающей функции, возможно **main**.

В то время как выполняется шаг рекурсии, первоначальное обращение к функции остается открытым, т.е. его выполнение не завершается. Шаг рекурсии может приводить к намного большему числу таких рекурсивных вызовов, которые продолжаются до тех пор, пока функция продолжает делить каждую последующую задачу, для решения которой она вызывалась, на две обобщенные части. Чтобы рекурсия в конце концов завершилась, функция каждый раз должна вызываться для решения все более простого варианта первоначальной задачи, и эта последовательность все меньших и меньших задач должна в конце концов, свестись к основному случаю. В этой точке функция распознает основной случай, возвращает результат предыдущей копии функции, и далее следует последовательность возвратов по обратному пути до тех пор, пока первоначальный вызов функции не вернет конечный результат в **main**. Все это звучит совершенно экзотически по сравнению со стандартными способами решения задач, которые мы применяли до сих пор. Действительно, для написания рекурсивных программ требуется большая практика, прежде чем рекурсия покажется чем-то естественным. Для иллюстрации работы этой концепции мы создадим рекурсивную программу, выполняющую распространенный вид математических вычислений.

Факториал целого неотрицательного числа n записывается как $n!$ (произносится «n факториал») и является произведением

$$n! = n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot 1$$

так что 1! равен 1, а 0! равен 1 по определению. Например, 5! является произведением 5 * 4 * 3 * 2 * 1, которое равно 120.

Факториал целого **number**, большего или равного 0, может быть вычислен *итеративно* (не рекурсивно) с помощью цикла **for**:

```
factorial = 1;
for (counter = number; counter >= 1; counter--)
   factorial *= counter;
```

Рекурсивное определение функции для вычисления факториала основано на следующем соотношении:

$$n! = n \cdot (n-1)!$$

Например, очевидно, 5! равен 5 * 4!, как показано ниже:

$$5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1$$
$$5! = 5 \cdot (4 \cdot 3 \cdot 2 \cdot 1)$$
$$5! = 5 \cdot (4!)$$

Вычисление 5! могло бы выполняться по схеме, приведенной на рис. 5.13. Рис. 5.13a показывает, как происходит последовательность рекурсивных вызовов до тех пор, пока не будет вычислено значение 1!, равное 1, которым заканчивается рекурсия. На рис. 5.13b показаны значения, возвращаемые вызывающей функции из каждого рекурсивного вызова. Возврат значений происходит до тех пор, пока не будет вычислено и возвращено конечное значение.

Программа, приведенная на рис. 5.14, использует рекурсию для вычисления и вывода на печать факториалов целых чисел от 0 до 10 (выбор типа данных **long** поясняется чуть ниже). Рекурсивная функция **factorial** сначала проверяет, выполняется ли условие завершения: переменная **number** меньше или равна 1. Если **number** действительно меньше или равна 1, **factorial** возвращает 1; в дальнейшей рекурсии необходимости нет и программа завершается. Если **number** больше 1, оператор

```
return number * factorial(number - 1);
```

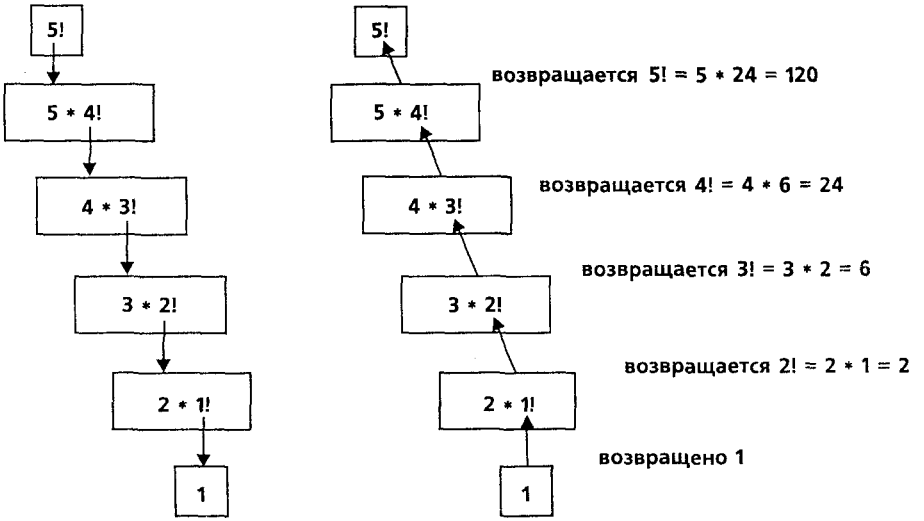


Рис. 5.13. Рекурсивное вычисление 5!

представляет задачу как вычисление произведения **number** на рекурсивный вызов **factorial**, возвращающий факториал от **number - 1**. Обратите внимание, что факториал от **(number - 1)** — немного более простая задача, чем исходная задача **factorial(number)**.

В объявлении функции **factorial** указано, что она получает параметр типа **long** и возвращает результат типа **long**. Эта запись является сокращением от **long int**. Стандарт ANSI определяет, что переменная типа **long int** хранится по меньшей мере в 4-х байтах и, следовательно, может иметь значения до +2147483647. Как можно видеть из рис. 5.14, значения факториала быстро возрастают. Мы выбрали тип данных **long** для того, чтобы программа могла вычислять значения факториалов больших 7! на компьютерах с малой разрядностью целых чисел (например, 2-байтовых). Для печати значений типа **long** используется спецификация преобразования **%ld**. К сожалению, функция **factorial** приобретает большие значения настолько быстро, что даже тип **long int** не позволяет напечатать большое число факториалов без превышения размера переменной типа **long int**.

Как мы еще увидим в упражнениях, пользователям, которым необходимо вычислять факториалы больших чисел, придется использовать типы **float** и **double**. Это указывает на слабость C (и большинства других языков программирования), а именно на то, что язык не так легко расширить, чтобы удовлетворить уникальные требования различных приложений. Как мы увидим позже, C++ является расширяемым языком, который, если мы того желаем, позволяет создавать произвольно большие целые числа.

Распространенная ошибка программирования 5.15

Отсутствие возврата значения из рекурсивной функции, когда он необходим.

```
/* Рекурсивная функция factorial */
#include <stdio.h>

long factorial(long);

main()
{
   int i;

   for (i = 1; i <= 10; i++)
      printf("%2d != %ld\n", i, factorial(i));

   return 0;
}

/* Рекурсивное определение функции factorial */
long factorial(long number)
{
   if (number <= 1)
      return 1;
   else
      return (number * factorial(number - 1));
}
```

```
 1! = 1
 2! = 2
 3! = 6
 4! = 24
 5! = 120
 6! = 720
 7! = 5040
 8! = 40320
 9! = 362880
10! = 3628800
```

Рис. 5.14. Вычисление факториалов рекурсивной функцией

Распространенная ошибка программирования 5.16

Пропуск основного случая или некорректная запись шага рекурсии приводит к тому, что рекурсия не сходится к основному случаю. Это вызывает бесконечную рекурсию и приводит, в конечном счете, к переполнению памяти. Аналогией этому является проблема бесконечного цикла в итеративном (нерекурсивном) решении. Бесконечная рекурсия может быть также вызвана неправильным вводом значения.

5.14. Пример применения рекурсии: числа Фибоначчи

Числа Фибоначчи

0,1,1,2,3,5,8,13,21, ...

начинаются с 0 и 1 и имеют то свойство, что каждое последующее число Фибоначчи равно сумме двух предыдущих.

Числа Фибоначчи встречаются в природе и, в частности, описывают форму спирали. Отношение последовательных чисел Фибоначчи сходится к значению константы 1.618.... Это число также неоднократно встречается в природе

и называется *золотым сечением* или *золотым средним*. Люди стремятся найти в золотом сечении эстетический аспект. Архитекторы часто разрабатывают окна, комнаты и здания, у которых отношение длины и ширины соответствует золотому сечению. Почтовые открытки также делаются с отношением сторон, близким к значению золотого сечения.

Числа Фибоначчи могут быть выражены при помощи рекурсии следующим образом:

```
fibonacci(0) = 0
fibonacci(1) = 1
fibonacci(n) = fibonacci(n - 1) + fibonacci(n - 2)
```

Программа, приведенная на рис. 5.15, вычисляет *i*-ое число Фибоначчи, вызывая рекурсивную функцию **fibonacci**. Обратите внимание, что числа Фибоначчи имеют тенденцию быстрого роста. Поэтому для параметра и возвращаемого значения функции **fibonacci** мы выбрали тип данных **long**. На рис. 5.15 каждая пара строк вывода показывает результат независимого запуска программы.

Вызов **fibonacci** из **main** не является рекурсивным вызовом, но все последующие вызовы **fibonacci** выполняются рекурсивно. Каждый раз когда вызывается **fibonacci**, следует немедленная проверка на основной случай, т.е. на равенство **n 0** или **1**. Если это условие выполнено, то следует возврат **n**. Интересно, что если **n** больше **1**, шаг рекурсии генерирует два рекурсивных вызова, каждый из которых решает немного более простую задачу, чем первоначальный вызов **fibonacci**. На рис. 5.16 показано, как функция **fibonacci** вычислила бы значение **fibonacci(3)** (мы сократили имя **fibonacci** до **f**, чтобы сделать рисунок более ясным).

Этот рисунок поднимает некоторые интересные вопросы, касающиеся порядка, в котором компиляторы С вычисляют значения операндов в операциях. Он не имеет отношения к тому порядку, в котором операции применяются к их операндам, т.е. от порядка, диктуемого правилами приоритета операций. Из рис. 5.16 видно, что при вычислении значения **f(3)** будет сделано два рекурсивных вызова: **f(2)** и **f(1)**. Но в каком порядке будут сделаны эти вызовы? Большинство программистов просто полагает, что значения операндов будут вычисляться слева направо. Странно, но стандарт ANSI не определяет порядок, в котором должны вычисляться значения операндов большинства операций (включая +). Следовательно, программист не может делать никаких предположений относительно порядка, в котором будут выполняться эти вызовы. Действительно, сначала мог бы следовать вызов **f(2)**, а затем **f(1)**, или вызовы могли бы выполняться в обратном порядке — **f(1)**, а затем **f(2)**. В этой программе и в большинстве других программ окажется, что конечный результат для обоих случаев был бы тем же самым. Но в некоторых программах вычисление значения операнда может иметь побочные эффекты, которые могли бы воздействовать на конечный результат выражения. Из большого числа операций С стандарт ANSI определяет порядок вычисления операндов только в четырех операциях: &&, ||, операции-запятой (,) и операции ?:. Первые три из них — это двухместные операции, два операнда которых гарантированно будут вычисляться слева направо. Последняя операция — это единственная трехместная операция. Ее крайний левый операнд всегда вычисляется в первую очередь; если крайний левый операнд дает результат, отличающийся от нуля, то вычисляется значение среднего операнда, а последний операнд игнорируется; если крайний левый операнд дает нуль, то вычисляется значение третьего операнда, а средний игнорируется.

```
/* Рекурсивная функция fibonacci */
#include <stdio.h>

long fibonacci(long);

main()
{
   long result, number;

   printf("Enter an integer: ");
   scanf("%ld", &number);
   result = fibonacci(number);
   printf("Fibonacci(%ld) = %ld\n", number, result);
   return 0;
}

/* Рекурсивное определение функции fibonacci */
long fibonacci(long n)
{
   if (n == 0 || n == 1)
      return n;
   else
      return fibonacci(n - 1) + fibonacci(n - 2);
}
```

```
Enter an integer: 0
Fibonacci(0) = 0

Enter an integer: 1
Fibonacci(1) = 1

Enter an integer: 2
Fibonacci(2) = 1

Enter an integer: 3
Fibonacci(3) = 2

Enter an integer: 4
Fibonacci(4) = 3

Enter an integer: 5
Fibonacci(5) = 5

Enter an integer: 6
Fibonacci(6) = 8

Enter an integer: 10
Fibonacci(10) = 55

Enter an integer: 20
Fibonacci(20) = 6765

Enter an integer: 30
Fibonacci(30) = 832040

Enter an integer: 35
Fibonacci(35) = 9227465
```

Рис. 5.15. Рекурсивная генерация чисел Фибоначчи.

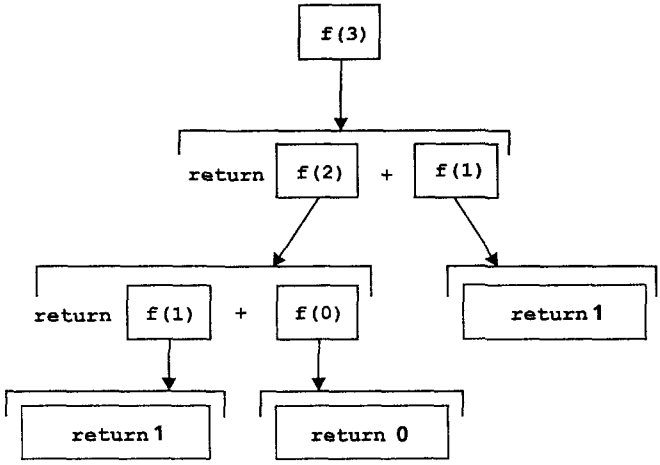


Рис. 5.16. Последовательность рекурсивных вызовов функции **fibonacci**

Распространенная ошибка программирования 5.17

Создание программы, которая зависит от порядка вычисления значений операндов операций, не являющихся **&&**, **||**, **?:** и операцией-запятой (,), может привести к ошибкам, поскольку компиляторы могут вычислять значения операндов не в том порядке, который ожидал программист.

Советы по переносимости программ 5.2

Программы, которые зависят от порядка вычисления значений операндов операций, не являющихся **&&**, **||**, **?:** и операцией-запятой (,), могут функционировать по-разному на системах с различными компиляторами.

Следует сказать несколько предостерегающих слов относительно рекурсивных программ, подобных той, что мы написали здесь для генерации чисел Фибоначчи. Каждый уровень рекурсии в функции **fibonacci** удваивает число вызовов, т.е. число рекурсивных вызовов, которые будут выполнены для вычисления числа Фибоначчи с номером **n**, растет как 2^n. Этот процесс быстро выходит из-под контроля. Вычисление только 20-го числа Фибоначчи потребовало бы порядка 2^{20}, или около миллиона вызовов; вычисление 30-го числа Фибоначчи требует порядка 2^{30}, или около миллиарда вызовов, и так далее. Ученые, работающие с компьютерами, называют эти задачи задачами с *экспоненциальной сложностью*. Эти задачи сложны даже для наиболее производительных компьютеров, имеющихся на настоящий момент! Проблемы сложности задачи вообще, и экспоненциальной сложности в частности, подробно обсуждаются в курсе по компьютерным дисциплинам, который называется «Алгоритмы».

Совет по повышению эффективности 5.4

Избегайте рекурсивных программ, подобных вычислению чисел Фибоначчи, которые приводят к экспоненциальному «взрыву» рекурсивных вызовов.

5.15. Рекурсия в сравнении с итерацией

В предыдущих разделах мы исследовали две функции, которые легко могут быть реализованы посредством как рекурсии, так и итерации. В этом разделе мы сравним эти два подхода и обсудим, почему программист может предпочесть один подход другому в конкретном приложении.

И итерация и рекурсия основаны на управляющих структурах: итерация использует структуру повторения; рекурсия использует структуру выбора. И итерация и рекурсия подразумевают повторение: итерация использует структуру повторения явным образом; рекурсия — посредством повторных вызовов функции. Итерация и рекурсия включают проверку на завершение: итерация завершается, когда перестает выполняться условие продолжения цикла; рекурсия завершается, когда распознается основной случай. Как итерация, с ее проверкой повторения по состоянию счетчика, так и рекурсия приближаются к завершению постепенно: итерация продолжает изменять счетчик, пока счетчик не примет значение, которое перестает удовлетворять условию продолжения цикла; рекурсия продолжает производить более простые варианты первоначальной задачи, пока не будет достигнут основной случай. И итерация и рекурсия может происходить бесконечно: итерация попадает в бесконечный цикл, если условие продолжения цикла никогда не становится ложным; рекурсия продолжается бесконечно, если шаг рекурсии не редуцирует задачу таким образом, что задача сходится к основному случаю.

Рекурсия имеет много отрицательных сторон. Она многократно инициирует механизм вызова функций и, соответственно, увеличивает связанные с ним накладные расходы. Это может дорого стоить как в плане процессорного времени, так и в плане расхода памяти. Каждое рекурсивное обращение создает другую копию функции (фактически только копию ее переменных), и это может расходовать значительные ресурсы памяти. Итерация обычно происходит в пределах функции, так что здесь нет накладных расходов на повторные вызовы функций и дополнительное выделение памяти. Так по какой причине для решения задачи можно выбрать рекурсию?

Общее методическое замечание 5.12

Любая проблема, которая может быть решена рекурсивно, может также быть решена итеративно (не рекурсивно). Рекурсивный подход обычно предпочитается итеративному в тех случаях, когда рекурсия более естественно отражает математическую сторону задачи и приводит к программе, которая проще для понимания и отладки. Другой причиной для выбора рекурсивного решения является то, что итеративное решение может не быть очевидным.

Совет по повышению эффективности 5.5

Избегайте использовать рекурсию в ситуациях, требующих производительности программы. Рекурсивные обращения требуют времени и расходуют дополнительную память.

Распространенная ошибка программирования 5.18

Случайный вызов нерекурсивной функцией самой себя или непосредственно, или косвенно через другую функцию.

Большинство учебников по программированию представляет рекурсию намного позже, чем сделано здесь. Мы считаем, что рекурсия — это ценный и сложный предмет, который лучше вводить раньше и рассматривать примеры рекурсии по всему остальному тексту. На рис. 5.17 перечислен 31 пример рекурсии, представленный в различных главах книги.

Позвольте нам закончить эту главу некоторыми замечаниями, которые мы неоднократно делаем во всей книге. Систематизация разработки программного обеспечения важна. Высокая производительность программ также важна. К сожалению, эти цели часто противоречат друг другу. Систематическое конструирование программного обеспечения является ключом к более эффективному управлению процессом создания еще больших и более сложных систем. Высокая производительность является ключом к реализации систем будущего, которые когда-нибудь предъявят еще большие требования к аппаратным средствам. Какое же отношение все это имеет к функциям?

Глава	Примеры и упражнения по рекурсии
Глава 5	Функция factorial Функция fibonacci Наибольший общий делитель Сумма двух целых Произведение двух целых Возведение целого в целую степень Ханойская башня Рекурсия **main** Печать ввода с клавиатуры в обратном порядке Визуализация рекурсии
Глава 6	Сумма элементов массива Печать массива Печать массива в обратном порядке Печать строки в обратном порядке Проверка, является ли строка палиндромом Минимальное значение в массиве Сортировка выбором Быстрая сортировка Линейный поиск Двоичный поиск
Глава 7	Восемь ферзей Обход лабиринта
Глава 8	Печать строки, введенной с клавиатуры, в обратном порядке
Глава 12	Вставка в связанный список Удаление из связанного списка Поиск в связанном списке Печать связанного списка в обратном порядке Вставка в двоичное дерево Обход двоичного дерева с предварительной выборкой Обход двоичного дерева с порядковой выборкой Обход двоичного дерева с отложенной выборкой

Рис. 5.17. Список примеров и упражнений по рекурсии, приведенных в тексте

Общее методическое замечание 5.13

Разбиение программы на функции аккуратным, иерархическим образом способствует систематизации конструирования программного обеспечения. Но за это приходится платить.

Совет по повышению эффективности 5.6

Программа, интенсивно использующая функции, по сравнению с монолитной (т.е. состоящей из одной части) программой без функций имеет потенциально большое число вызовов, а это увеличивает процессорное время программы. Но монолитные программы трудны для написания, тестирования, отладки, сопровождения и развития.

Таким образом, создание вашей программы из многих функций является разумным решением, но всегда нужно учитывать шаткое равновесие между производительностью и стремлением к систематическому конструированию программного обеспечения.

Резюме

- Наилучшим способом разработки и сопровождения больших программ является создание программы из небольших программных модулей, каждый из которых проще в обращении, чем исходная программа. В C *модули реализуются как функции.*
- Обращение к функции осуществляется посредством вызова. В вызове функции указывается ее имя и передается информация (в виде аргументов), необходимая функции для выполнения своей задачи.
- Целью сокрытия информации является стремление к тому, чтобы функции имели доступ только к той информации, которая им необходима для выполнения своих задач. Смысл этого состоит в применении принципа минимизации привилегий — одного из наиболее важных принципов систематической разработки программного обеспечения.
- Функции обычно вызываются в программе посредством записи имени функции с последующими левой круглой скобкой, аргументом функции (или списком аргументов, отделяемых запятыми) и правой круглой скобкой.
- Данные типа **double** являются числами с плавающей точкой, аналогичными **float**. Переменные типа **double** могут хранить значения намного большей величины и точности, чем **float**.
- Любой аргумент функции может быть константой, переменной или выражением.
- Локальная переменная известна только той функции, где она определена. Другим функциям не разрешается знать имена локальных переменных данной функции, равно как и любой функции не разрешено знать подробности реализации других функций.
- Общий формат определения функции имеет следующий вид:

```
тип_возвращаемого_значения имя_функции(список_параметров)
{
объявления
операторы
}
```

Тип_возвращаемого_значения объявляет тип значения, возвращаемого вызывающей функции. Если функция не возвращает значения, то *тип_возвращаемого_значения* объявляется как **void**. *Имя_функции* —

это любой допустимый идентификатор. *Список_параметров* — это список объявлений параметров (разделенных запятыми), получаемых функцией при ее вызове. Если функция не получает значений, список параметров объявляется как **void**. *Тело функции* — набор объявлений и операторов, которые составляют собственно функцию.

- Аргументы, переданные функции, должны соответствовать по числу, типу и порядку следования параметрам, указанным в определении функции.
- Когда программа встречает имя функции, то управление из точки вызова передается вызываемой функции, выполняются операторы вызываемой функции и управление возвращается структуре, вызвавшей функцию.
- Вызываемая функция может вернуть управление вызывающему оператору одним из трех способов. Если функция не возвращает значения, управление возвращается, когда достигается правая фигурная скобка, завершающая тело функции, или посредством оператора

  ```
  return;
  ```

  Если функция возвращает значение, оператор

  ```
  return expression;
  ```

  возвращает значение выражения **expression.**
- Прототип функции объявляет тип, возвращаемый функцией, а также число, типы и порядок следования параметров, которые она должна получить.
- Прототипы дают возможность компилятору проверить, что функция вызвана правильно.
- Компилятор игнорирует имена переменных, указанных в прототипе функции.
- Каждая стандартная библиотека имеет соответствующий заголовочный файл, содержащий прототипы для всех функций этой библиотеки, а также содержит определения различных символических констант, необходимых этим функциям.
- Программисты могут создавать и включать в программу свои собственные заголовочные файлы.
- Когда аргумент передается по значению, значение переменной *копируется* и копия передается вызываемой функции. Изменения копии в вызываемой функции не отражаются на значении первоначальной переменной.
- Все вызовы в C передают аргументы по значению.
- Функция **rand** генерирует целое значение в диапазоне от **0** до **RAND_MAX**. Стандарт ANSI устанавливает, что значение **RAND_MAX** должно быть равно по крайней мере 32767.
- Прототипы функций **rand** и **srand** содержатся в **<stdlib.h>**.
- Значения, генерируемые **rand**, могут масштабироваться и сдвигаться для достижения необходимого диапазона значений.

- Для рандомизации программы используйте функцию **srand** из стандартной библиотеки С.
- Оператор с функцией **srand** обычно вставляется в программу только после того, как программа полностью отлажена. При отладке лучше всего опустить **srand**. Это гарантирует повторяемость результатов, которая существенна для доказательства того, что исправления, внесенные в программу с генератором случайных чисел работают правильно.
- Чтобы рандомизировать программу, не вводя каждый раз новые значения семени, можно использовать **srand (time (NULL))**. Функция **time** возвращает число секунд, прошедшее с начала дня. Прототип функции **time** находится в заголовочном файле **<time.h>**.
- Общей формулой для масштабирования и сдвига случайных чисел является

  ```
  n = a + rand() % b;
  ```

  где **a** — значение смещения (которое равно первому числу задаваемого диапазона последовательных целых чисел), и **b** — коэффициент масштабирования (который равен ширине задаваемого диапазона последовательных целых чисел).
- Каждый идентификатор в программе имеет атрибуты класса памяти: период хранения, область действия и тип компоновки.
- С поддерживает четыре класса памяти, обозначаемые спецификаторами класса памяти: ***auto***, ***register***, ***extern*** и ***static***.
- Период хранения идентификатора — это время, в течение которого данный идентификатор существует в памяти.
- Область действия идентификатора характеризует возможность обращения к нему из различных частей программы.
- Тип компоновки идентификатора определяется для программ, состоящих из нескольких исходных файлов. Эта характеристика показывает, известен ли идентификатор только в текущем исходном файле или в любом исходном файле с соответствующими объявлениями.
- Переменные с автоматическим периодом хранения создаются при запуске блока, в котором они объявлены, существуют, пока блок активен, и разрушаются, когда блок завершает работу. Локальные переменные функции обычно имеют автоматический период хранения.
- Спецификатор класса памяти **register** может быть помещен перед объявлением автоматической локальной переменной, чтобы рекомендовать компилятору разместить переменную в одном из быстродействующих аппаратных регистров процессора. Компилятор может игнорировать объявление **register**. Ключевое слово **register** может использоваться только с переменными, имеющими автоматический период хранения.
- Ключевые слова **extern** и **static** используются при объявлении идентификаторов переменных и функций со статическим периодом хранения.
- Переменные со статическим периодом хранения размещаются и инициализируются один раз, когда программа начинает выполняться.

- Существует два типа идентификаторов со статическим периодом хранения: внешние идентификаторы (такие как глобальные переменные и имена функций) и локальные переменные, объявляемые со спецификатором класса памяти **static**.
- Глобальные переменные создаются при объявлении переменных вне любого определения функции, и они сохраняют свои значения в течение всего времени выполнения программы.
- Локальные переменные, объявленные как **static**, сохраняют свои значения и после того, как функция, в которой они объявляются, завершает работу.
- Все числовые переменные со статическим периодом хранения инициализируются обнулением, если они не инициализированы программистом явным образом.
- Четырьмя областями действия для идентификаторов являются: область действия функции, область действия файла, область действия блока и область действия прототипа функции.
- Метки — это единственные идентификаторы с областью действия функции. На метки можно ссылаться в любом месте функции, в которой они появляются, но вне тела функции на них ссылаться нельзя.
- Идентификатор, объявленный вне любой функции, имеет область действия файла. Такой идентификатор «известен» во всех функциях начиная с места, в котором идентификатор был объявлен, и до конца файла.
- Идентификаторы, объявленные внутри блока, имеют область действия блока. Область действия блока заканчивается правой фигурной скобкой (}), завершающей блок.
- Локальные переменные, объявленные в начале функции, имеют область действия блока, так же как и параметры функции, которые рассматриваются в качестве ее локальных переменных.
- Любой блок может содержать объявления переменных. Когда блоки вложены, а идентификатор во внешнем блоке имеет то же самое имя, что и идентификатор во внутреннем блоке, идентификатор во внешнем блоке оказывается «скрытым», пока внутренний блок не завершит работу.
- Единственными идентификаторами с областью действия прототипа функции являются идентификаторы, которые встречаются в списке параметров прототипа. Идентификаторы, указанные в прототипе функции, могут неоднократно использоваться в других местах программы, при этом не возникает никакой неоднозначности.
- Рекурсивная функция — это функция, которая вызывает саму себя или непосредственно, или косвенно.
- Если функция вызывается для решения основного случая, то она просто возвращает результат. Если функция вызывается для решения более сложной задачи, функция делит задачу на две обобщенных части: часть, для которой функция имеет способ решения, и часть, для которой функция решения не имеет. Поскольку эта новая задача похожа на исходную, функция запускает (вызывает) свою новую копию, чтобы продолжить решение редуцированной задачи.

- Чтобы рекурсия завершилась, функция каждый раз должна вызываться для решения немного более простого варианта исходной задачи, и эта последовательность все меньших и меньших задач должна, в конечном счете, свестись к основному случаю. Когда функция распознает основной случай, она возвращает результат предыдущей копии функции и далее следует последовательность возвратов по обратному пути до тех пор, пока первоначальный вызов функции не вернет конечный результат.
- Стандарт ANSI не определяет порядок, в котором должны вычисляться значения операндов большинства операций (включая +). Из большого числа операций C стандарт ANSI определяет порядок вычисления операндов только в четырех операциях: &&, ||, операции-запятой (,) и операции ?:. Первые три из них — это двуместные операции, два операнда которых гарантированно будут вычисляться слева направо. Последняя операция — единственная трехместная операция. Ее крайний левый операнд всегда вычисляется в первую очередь; если крайний левый операнд дает результат, отличающийся от нуля, то вычисляется значение среднего операнда, а последний операнд игнорируется; если крайний левый операнд дает нуль, то вычисляется значение третьего операнда, а средний операнд игнорируется.
- И итерация и рекурсия основаны на управляющих структурах: итерация использует структуру повторения; рекурсия использует структуру выбора.
- И итерация и рекурсия подразумевают повторение: итерация использует структуру повторения явным образом; рекурсия — посредством повторных вызовов функции.
- Итерация и рекурсия включают проверку на завершение: итерация завершается, когда перестает выполняться условие продолжения цикла; рекурсия завершается, когда распознается основной случай.
- И итерация и рекурсия может происходить бесконечно: итерация попадает в бесконечный цикл, если условие продолжения цикла никогда не становится ложным; рекурсия продолжается бесконечно, если шаг рекурсии не редуцирует задачу таким образом, что она сходится к основному случаю.
- Рекурсия многократно инициирует механизм вызова функций и, соответственно, увеличивает связанные с ним накладные расходы. Это может дорого стоить как в плане процессорного времени, так и в плане расхода памяти.

Терминология

clock
rand
RAND_MAX
return
srand
time
unsigned
void
абстракция
автоматическая переменная
автоматический класс памяти
автоматическое приведение аргументов
активация функции
аргумент в вызове функции
блок
вызов по значению
вызов по ссылке

вызов функции
вызываемая функция
вызывающая функция
выражение со смешанными типами
генерация случайных чисел
глобальная переменная
заголовочные файлы стандартной библиотеки
заголовочный файл
иерархия возведения
итерация
классы памяти
конструирование программного обеспечения
копия значения
локальная переменная
масштабирование
моделирование
модульная программа
область действия
область действия блока
область действия прототипа
область действия файла
область действия функции
объявление функции
определение функции
оптимизирующий компилятор
основной случай рекурсии
параметр в определении функции
переменная класса static
период хранения
побочные эффекты
повторное использование программного кода
принцип минимума привилегий
прототип функции
псевдослучайные числа
«разделяй и властвуй»
рандомизация
рекурсивная функция
рекурсивный вызов
рекурсия
сдвиг
сокрытие информации
спецификатор класса памяти
спецификатор класса памяти auto
спецификатор класса памяти extern
спецификатор класса памяти register
спецификатор класса памяти static
спецификатор преобразования %s
стандартная библиотека C
статический период хранения
тип возвращаемого значения
тип компоновки
функции математической библиотеки
функция
функция факториала
функция, определяемая программистом
элемент случайности

Распространенные ошибки программирования

5.1. При отсутствии в программе заголовочного файла математики обращение программы к функциям математической библиотеки может приводить к странным результатам.

5.2. Если *тип_возвращаемого_значения* в определении функции опущен, то это вызовет синтаксическую ошибку в том случае, если прототип функции определяет, что возвращаемый тип не является целым (**int**).

5.3. Если функция должна возвращать некоторое значение, а в функции опущен оператор возврата, то это может привести к непредвиденным ошибкам. Стандарт ANSI гласит, что в этом случае результат не определен.

5.4. Возврат значения из функции, для которой возвращаемый тип был объявлен как **void**, вызывает синтаксическую ошибку.

5.5. Запись **float x, y** вместо **float x, float y** при объявлении параметров функции, относящихся к одному типу. Такое объявление параметров, как **float x, y** фактически присвоило бы параметру **y** тип **int**, поскольку **int** принимается по умолчанию.

5.6. Символ точки с запятой после правой круглой скобки, закрывающей список параметров в определении функции, является синтаксической ошибкой.

5.7. Повторное определение параметра функции как локальной переменной внутри функции — синтаксическая ошибка.

5.8. Определение функции внутри другой функции — синтаксическая ошибка.

5.9. Пропуск точки с запятой в конце прототипа функции вызывает синтаксическую ошибку.

5.10. Преобразование более высокого типа данных в иерархии возведения к более низкому типу может изменить значение данных.

5.11. Пропуск прототипа функции вызывает синтаксическую ошибку, если функция возвращает тип, отличный от **int**, а определение функции появляется в программе после вызова функции. В других случаях пропуск прототипа функции может вызвать ошибку во время выполнения программы или привести к непредвиденному результату.

5.12. Использование **srand** вместо **rand** для генерации случайных чисел.

5.13. Использование нескольких спецификаторов класса памяти для идентификатора. К данному идентификатору может применяться только один спецификатор класса памяти.

5.14. *Случайное объявление во внутреннем блоке имени идентификатора*, которое уже используется во внешнем блоке, в то время как программист на самом деле хотел, чтобы идентификатор внешнего блока был доступен для внутреннего блока.

5.15. Отсутствие возврата значения из рекурсивной функции, когда он необходим.

5.16. Пропуск основного случая или некорректная запись шага рекурсии приводит к тому, что рекурсия не сходится на основном случае. Это вызывает бесконечную рекурсию и приводит, в конечном счете, к переполнению памяти. Аналогией этому является проблема бесконечного цикла в итеративном (не рекурсивном) решении. Бесконечная рекурсия может быть также вызвана неправильным вводом значения.

5.17. Создание программы, которая зависит от порядка вычисления значений операндов операций, не являющихся &&, , ?: и операцией-запятой (,), может привести к ошибкам, поскольку компиляторы могут вычислять значения операндов не в том порядке, который ожидал программист.

5.18. Случайный вызов нерекурсивной функцией самой себя или непосредственно, или косвенно через другую функцию.

Хороший стиль программирования

5.1. Освойтесь с широким набором функций стандартной библиотеки ANSI C.

5.2. При использовании функций математической библиотеки включите в программу заголовочный файл математики с помощью директивы препроцессора **#include <math.h>**.

5.3. Вставляйте пустую строку между определениями функций, чтобы отделить функции и улучшить тем самым читаемость программы.

5.4. Хотя опущенный возвращаемый тип по умолчанию рассматривается как **int**, всегда задавайте возвращаемый тип явным образом. Однако для функции **main** возвращаемый тип обычно опускается.

5.5. Включайте тип каждого параметра в список параметров функции даже в том случае, если данный параметр относится к типу **int**, принятому по умолчанию.

5.6. Хотя это и не является ошибкой, не используйте одни и те же имена для аргументов, переданных функции, и соответствующих параметров в определении функции. Это поможет избежать неоднозначности.

5.7. Выбор осмысленных имен функций и параметров делает программы более удобочитаемыми и помогает избежать чрезмерного использования комментариев.

5.8. Включите в программу прототипы функций для всех функций, чтобы воспользоваться преимуществами проверки типов С. Используйте директивы препроцессора **#include**, чтобы получить прототипы функций стандартной библиотеки из заголовочных файлов соответствующих библиотек. Также используйте **#include**, чтобы включить заголовочные файлы, содержащие ваши собственные прототипы функций или прототипы, используемые членами вашей рабочей группы.

5.9. Имена параметров иногда включаются в прототип функции с целью документирования. Компилятор игнорирует эти имена.

5.10. Переменные, использующиеся только в определенной функции, должны быть объявлены локальными переменными этой функции, а не внешними переменными.

5.11. Не объявляйте имен переменных, которые скрывают имена во внешних областях действия. Можно просто избегать любого дублирования имен идентификаторов в программе.

Советы по переносимости программ

5.1. Использование функций стандартной библиотеки ANSI С делает программы более переносимыми.

5.2. Программы, которые зависят от порядка вычисления значений операндов операций, не являющихся &&, , ?: и операцией-запятой (,), могут функционировать по-разному на системах с различными компиляторами.

Советы по повышению эффективности

5.1. Автоматическое хранение способствует экономии памяти, поскольку автоматические переменные существуют только тогда, когда они необходимы. Они создаются при запуске функции, в которой они объявлены, и уничтожаются, когда происходит выход из функции.

5.2. Перед объявлением автоматической переменной может быть помещен спецификатор класса памяти **register**, чтобы рекомендовать компилятору разместить ее в одном из быстродействующих аппаратных регистров компьютера. Если интенсивно используемые переменные типа счетчиков или сумм будут реализованы в аппаратных регистрах, то можно исключить непроизводительные затраты на неоднократную загрузку переменных из памяти в регистры и обратно.

5.3. Объявления **register** часто бывают не нужны. Современные оптимизирующие компиляторы способны распознать часто используемые переменные и могут размещать их в регистрах самостоятельно, не требуя от программиста объявления **register**.

5.4. Избегайте рекурсивных программ, подобных вычислению чисел Фибоначчи, которые приводят к экспоненциальному «взрыву» рекурсивных вызовов.

5.5. Избегайте использовать рекурсию в ситуациях, требующих производительности программы. Рекурсивные обращения требуют времени и расходуют дополнительную память.

5.6. Программа, интенсивно использующая функции, по сравнению с монолитной (т.е. состоящей из одной части) программой без функций имеет потенциально большое число вызовов, а это увеличивает процессорное время программы. Но монолитные программы трудны для написания, тестирования, отладки, сопровождения и развития.

Общие методические замечания

5.1. Не изобретайте колесо. Если есть возможность, используйте функции стандартной библиотеки ANSI C вместо того, чтобы создавать новые функции. Это уменьшает время разработки программы.

5.2. В программах, содержащих большое число функций, **main** должна быть реализована как группа обращений к функциям, выполняющим основную часть работы.

5.3. Каждая функция должна ограничиваться выполнением одной, точно определенной задачи, а имя функции должно отражать смысл данной задачи. Это облегчает абстракцию и поддерживает способствует многократному использованию программного кода.

5.4. Если вы не можете выбрать краткое имя, которое выражает назначение функции, возможно, что ваша функция пытается выполнить слишком много задач. Лучше разбить такую функцию на несколько меньших функций.

5.5. Длина функции не должна превышать одной страницы. Еще лучше, чтобы функция не превышала по длине половины страницы. Небольшие функции способствуют повторному использованию программного кода.

5.6. Программы должны составляться в виде совокупности небольших функций. Это упрощает создание программ, их отладку, поддержку и модификацию.

5.7. Функция, требующая большого количества параметров, может выполнять слишком много задач. Рассмотрите возможность деления функции на меньшие функции, которые выполняют выделенные задачи. Заголовок функции должен, по возможности, умещаться на одной строке.

5.8. Прототип функции, заголовок функции и вызов функции должны иметь взаимное соответствие по числу, типу и порядку следования аргументов и параметров, а также по типу возвращаемого значения.

5.9. Прототип функции, размещенный вне любого определения функции, применяется ко всем вызовам, появляющимся в файле после прототипа функции. Прототип функции, помещенный в функцию, применяется только к тем вызовам, которые делаются из этой функции.

5.10 Автоматическая хранение является примером реализации принципа минимальных привилегий. Почему переменные должны храниться в памяти и быть доступными, когда в данный момент в них нет необходимости?

5.11. Объявление переменной как глобальной, а не локальной, может приводить к случайным побочным эффектам, когда функция, которой не нужен доступ к этой переменной, случайно или преднамеренно ее изменяет. В целом нужно избегать использования глобальных переменных, за исключением некоторых ситуаций с уникальными требованиями к производительности (рассматривается в главе 14).

5.12. Любая проблема, которая может быть решена рекурсивно, может также быть решена итеративно (не рекурсивно). Рекурсивный подход обычно предпочитается итеративному в тех случаях, когда рекурсия более естественно отражает математическую сторону задачи и приводит к программе, которая проще для понимания и отладки. Другой причиной для выбора рекурсивного решения является то, что итеративное решение может не быть очевидным.

5.13. Разбиение программы на функции аккуратным, иерархическим образом способствует систематизации конструирования программного обеспечения. Но за все приходится платить.

Упражнения для самоконтроля

5.1. Ответьте на следующие вопросы:

a) Модуль программы на C называется _____.

b) Обращение к функция осуществляется посредством _____.

c) Переменная, которая известна только внутри функции, в которой она определена, называется _____.

d) Оператор _____ в вызываемой функции используется для передачи значения выражения вызывающей функции.

e) Ключевое слово _____ используется в заголовке функции, чтобы показать, что функция не возвращает значения или не содержит никаких параметров.

f) _____ идентификатора является частью программы, в которой идентификатор может быть использован.

g) Тремя способами возврата управления от вызываемой функции к вызывающей являются _____, _____ и _____.

h) _____ позволяет компилятору проверять число, типы и порядок следования аргументов, переданных функции.

i) Функция _____ используется для генерации случайных чисел.

j) Функция _____ устанавливает семя генератора случайных чисел для рандомизации программы.

k) Спецификаторами класса памяти являются _____, _____, _____ и _____.

l) Предполагается, что переменные, объявленные в блоке или в списке параметров функции, имеют _____ класс памяти, если не указано иначе.

m) Спецификатор класса памяти _____ рекомендует компилятору разместить переменную в одном из регистров компьютера.

n) Переменная, объявленная вне любого блока или функции, является _____ переменной.

o) Чтобы локальная переменная в функции сохраняла свое значение между вызовами функции, она должна быть объявлена со спецификатором класса памяти _____.

p) Четырьмя возможными областями действия идентификатора являются _____, _____, _____ и _____.

q) Функция, которая вызывает саму себя или непосредственно или косвенно является _____ функцией.

r) Рекурсивная функция обычно имеет два компонента: один компонент задает условие завершения рекурсии, проверяя текущую задачу на _____ случай, а второй компонент упрощает задачу для выполнения очередного рекурсивного вызова.

5.2. Для программы, приведенной ниже, установите область действия каждого из перечисленных элементов (область действия функции, область действия файла, область действия блока или область действия прототипа функции).

a) Переменная **x** в **main**.

b) Переменная **y** в **cube**.

c) Функция **cube**.

d) Функция **main**.

e) Прототип функции для **cube**.

f) Идентификатор y в прототипе функции **cube**.

```
#include <stdio.h>
int cube(int y);

main ()
{
   int x;

   for (x = 1; x <= 10; x++)
      printf("%d\n", cube(x));
}

int cube(int y)
{
   return y = y * y;
}
```

5.3. Напишите программу, которая проверяет, действительно ли примеры вызова функций математической библиотеки, показанные на рис. 5.2, дают приведенные на рисунке результаты.

5.4. Приведите заголовок функции для следующих функций.

a) Функция **hypotenuse (гипотенуза)**, которая получает два аргумента **side1** и **side2**, имеющих тип с плавающей точкой удвоенной точности, и которая возвращает результат в виде значения с плавающей точкой удвоенной точности.

b) Функция **smallest (наименьшее)**, которая получает три целых **x**, **y**, **z** и возвращает целое значение.

c) Функция **instructions (инструкции)**, которая не получает аргументов и ничего не возвращает (Примечание: такие функции часто используются для вывода на экран инструкций для пользователя.)

d) Функция **intToFloat**, которая получает целый аргумент **number** и возвращает результат типа с плавающей точкой.

5.5. Приведите прототипы следующих функций:

a) Функции, рассмотренной в упражнении 5.4a.

b) Функции, рассмотренной в упражнении 5.4b.

c) Функции, рассмотренной в упражнении 5.4c.

d) Функции, рассмотренной в упражнении 5.4d.

5.6. Напишите следующие объявления:

a) Целого **count**, которое должно обрабатываться в регистре. Инициализируйте **count** значением **0**.

b) Переменной с плавающей точкой **lastVal**, которая сохраняет свое значение между вызовами функции, в которой она определена.

c) Внешнего целого **number**, чья область действия должна ограничиваться оставшейся частью файла, в котором определена переменная.

5.7. Найдите ошибку в каждом фрагменте программы и объясните, как ее исправить (см. также упражнение 5.50):

a)
```
int g(void) {
   printf("Inside function g\n");

   int h(void) {
      printf("Inside function h\n");
   }
}
```

b)
```
int sum(int x, int y) {
   int result;
   result = x + y;
}
```

c)
```
int sum(int n) {
   if (n == 0)
      return 0;
   else
      n + sum(n - 1);
}
```

d)
```
void f(float a); {
   float a;

   printf("%f", a);
}
```

e)
```
void product(void) {
   int a, b, c, result;
   printf("Enter three integers: ")
   scanf ("%d%d%d", &a, &b, &c);
   result = a * b * c;
   printf("Result is %d", result);
   return result;
}
```

Ответы на упражнения для самоконтроля

5.1. a) функцией. b) вызова функции. c) локальной переменной. d) **return.** e) **void.** f) Область действия. g) **return**; или **return expression;** или достижение правой фигурной скобки, завершающей функцию. h) Прототип функции. i) **rand.** j) **srand.** k) **auto, register, extern, static.** l) автоматический m) **register.** n) внешней, глобальной. o) **static.** p) область действия функции, область действия файла, область действия блока, область действия прототипа. q) рекурсивной. r) основной.

5.2. a) область действия блока. b) область действия блока. c) область действия файла. d) область действия файла. e) область действия файла. f) область действия прототипа.

5.3.
```
/* Проверка функций математической библиотеки */
#include <stdio.h>
#include <math.h>

main( )
{
   printf("sqrt(%.1f) = %.1f\n", 900.0, sqrt(900.0));
   printf("sqrt(%.1f) = %.1f\n", 9.0, sqrt(9.0));
```

```
   printf("exp(%.1f) = %f\n", 1.0, exp(1.0));
   printf("exp(%.1f) = %f\n", 2.0, exp(2.0));
   printf("log(%f) = %.1f\n", 2.718282, log(2.718282));
   printf("log(%f) = %.1f\n", 7.389056, log(7.389056));
   printf("log10(%.1f) = %.1f\n", 1.0, log10(1.0));
   printf("log10(%.1f) = %.1f\n", 10.0, log10(10.0));
   printf("log10(%.1f) = %.1f\n", 100.0, log10(100.0));
   printf("fabs(%.1f) = %.1f\n", 13.5, fabs(13.5));
   printf("fabs(%.1f) = %.1f\n", 0.0, fabs(0.0));
   printf("fabs(%.1f) = %.1f\n", -13.5, fabs(-13.5));
   printf("ceil(%.1f) = %.1f\n", 9.2, ceil(9.2));
   printf("ceil(%.1f) = %.1f\n", -9.8, ceil(-9.8));
   printf("floor(%.1f) = %.1f\n", 9.2, floor(9.2));
   printf("floor(%.1f) = %.1f\n", -9.8, floor(-9.8));
   printf("pow(%.1f, %.1f) = %.1f\n",
      2.0, 7.0, pow(2.0, 7.0));
   printf("pow(%.1f, %.1f) = %.1f\n",
      9.0, 0.5, pow(9.0, 0.5));
   printf("fmod(%.3f/%.3f) = %.3f\n",
      13.675, 2.333, fmod(13.675, 2.333));
   printf("sin(%.1f) = %.1f\n", 0.0, sin(0.0));
   printf("cos(%.1f) = %.1f\n", 0.0, cos(0.0));
   printf("tan(%.1f) = %.1f\n", 0.0, tan(0.0));
}
```

```
sqrt(900.0) = 30.0
sqrt(9.0) = 3.0
exp(1.0) = 2.718282
exp(2.0) = 7.389056
log(2.718282) = 1.0
log(7.389056) = 2.0
log10(1.0) = 0.0
log10(10.0) = 1.0
log10(100.0) = 2.0
fabs(13.5) = 13.5
fabs(0.0) = 0.0
fabs(-13.5) = 13.5
ceil(9.2) = 10.0
ceil(-9.8) = -9.0
floor(9.2) = 9.0
floor(-9.8) = -10.0
pow(2.0, 7.0) = 128.0
pow(9.0, 0.5) = 3.0
fmod(13.675/2.333) = 2.010
sin(0.0) = 0.0
cos(0.0) = 1.0
tan(0.0) = 0.0
```

5.4. a) `double hypotenuse(double side1, double side2)`

b) `int smallest(int x, int y, int z)`

c) `void instructions(void)`

d) *`float intToFloat(int number)`*

5.5. a) `double hypotenuse(double, double);`

b) `int smallest(int, int, int);`

c) `void instructions(void);`

d) `float intToFloat(int);`

5.6. a) `register int count = 0;`

b) `static float lastVal;`

c) `static int number;`

Примечание: эти объявления должны находиться вне любого определения функции.

5.7. a) Ошибка: функция **h** определена в функции **g**.

Исправление: поместите определение **h** вне определения **g**.

b) Ошибка: эта функция, как предполагается, возвращает целое число, но этого нет.

Исправление: удалите переменную **result** и вставьте в функцию следующий оператор:

```
return x + y;
```

c) Ошибка: не возвращается результат **n + sum(n-1)**; **sum** возвращает неверный результат.

Исправление: перепишите оператор в условии **else** как

```
return n + sum (n - 1);
```

d) Ошибка: точка с запятой после правой круглой скобки, которая включает список параметров, и переопределение параметра a в определении функции.

Исправление: удалите точку с запятой после правой круглой скобки списка параметров и удалите объявление **float a;**.

e) Ошибка: функция возвращает значение, когда этого не предполагается.

Исправление: удалите оператор **return**.

Упражнения

5.8. Приведите значение x после выполнения каждого оператора

a) `x = fabs(7.5)`

b) `x = floor(7.5)`

c) `x = fabs(0.0)`

d) `x = ceil(0.0)`

e) `x = fabs(-6.4)`

f) `x = ceil(-6.4)`

g) `x = ceil(-fabs(-8+floor(-5.5)))`

5.9. Гараж требует оплатить минимальный взнос в $2.00 для парковки машины на время, не большее трех часов. За каждый час времени (или за неполный час) сверх 3-х часов гараж требует доплаты $0.50 в час. Максимальная плата за любые 24 часа составляет $10.00. Предположим, что здесь нет автомобилей, запаркованных сразу на время, большее 24-х часов. Напишите программу, которая будет вы-

числять и печатать плату за парковку для 3-х водителей, которые вчера запарковали свои автомобили в этом гараже. Вы должны будете ввести время начала парковки для каждого заказчика. Ваша программа должна печатать результаты в аккуратном табличном формате и должна вычислять и печатать общую сумму вчерашних платежей. Программа должна использовать функцию calculateCharges для определения платы от каждого клиента. Программа должна выводить данные в следующем формате:

```
Car      Hours      Charge
1          1.5        2.00
2          4.0        2.50
3         24.0       10.00
TOTAL     29.5       14.50
```

5.10. Одним из применений функции **floor** является округление значения до ближайшего целого. Оператор

```
y = floor(x + .5);
```

округлит число **x** до ближайшего целого и присвоит результат **y**. Напишите программу, которая считывает несколько чисел и использует данный оператор для округления каждого из этих чисел до ближайшего целого. Для каждого обработанного числа напечатайте первоначальное значение и округленное число.

5.11. Функция **floor** может использоваться для округления числа до заданного десятичного знака. Оператор

```
y = floor(x * 10 + .5) / 10;
```

округляет **x** до десятых. Оператор

```
y = floor(x * 100 + .5) / 100;
```

округляет **x** до сотых. Напишите программу, которая определяет следующие четыре функции для округления числа **x**:

a) `roundToInteger(number) /* округлить до целого */`

b) `roundToTenths(number)/* округлить до десятых */`

c) `roundToHundreths(number)/* округлить до сотых */`

d) `roundToThousandths(number)/* округлить до тысячных */`

Для каждого считанного значения ваша программа должна печатать первоначальное значение, число, округленное до ближайшего целого, число, округленное до десятых, число, округленное до сотых и число, округленное до тысячных.

5.12. Ответьте на следующие вопросы.

a) Что означает «выбирать числа случайным образом»?

b) Почему для моделирования стохастических игр шанс необходима функция **rand**?

c) По какой причине вы рандомизировали бы программу с помощью функции **srand**? При каких обстоятельствах рандомизация нежелательна?

d) Почему часто возникает необходимость масштабировать и/или сдвигать значения, генерируемые функцией **rand**?

e) Чем полезен метод компьютерных имитаций ситуаций реального мира?

5.13. Напишите операторы, которые присваивают целые случайные числа переменной **n** в следующих диапазонах:

a) `1 ≤ n ≤ 2`

b) `1 ≤ n ≤ 100`

c) `0 ≤ n ≤ 9`

d) `1000 ≤ n ≤ 1112`

e) `-1 ≤ n ≤ 1`

f) `-3 ≤ n ≤ 11`

5.14. Для каждого из следующих наборов целых чисел напишите одиночный оператор, который будет печатать число из набора случайным образом.

a) `2, 4, 6, 8, 10.`

b) `3, 5, 7, 9, 11.`

c) `6, 10, 14, 18, 22.`

5.15. Определите функцию **hypotenuse**, которая вычисляет длину гипотенузы прямоугольного треугольника по двум другим сторонам. Используйте эту функцию в программе для определения длины гипотенузы треугольников, приведенных ниже. Функция должна получать два аргумента типа **double** и возвращать значение гипотенузы также типа **double**.

Треугольник	Сторона 1	Сторона 2
1	3.0	4.0
2	5.0	12.0
3	8.0	15.0

5.16. Напишите функцию **integerPower(base, exponent)**, которая возвращает значение

$base^{exponent}$

Например, **integerPower (3, 4) = 3 * 3 * 3 * 3**. Предположим, что **exponent** является положительным ненулевым целым, а **base** целым. Для управления вычислением функция **integerPower** должна применять цикл **for**. Не используйте никаких функций математической библиотеки.

5.17. Напишите функцию **multiple** для двух целых, которая определяет, кратно ли второе число первому. Функция должна получать два целых аргумента и возвращать **1** (true), если второе число кратно первому, и **0** (false) в противном случае. Используйте эту функцию в программе, которая вводит серию пар целых чисел.

5.18. Напишите программу, которая вводит последовательность целых чисел и передает их по одному функции **even**, которая использует

операцию деления по модулю, чтобы определить, является ли целое четным. Функция должна получать один аргумент и возвращать **1**, если целое четно и **0** в противном случае.

5.19. Напишите функцию, которая выводит у левой границы экрана сплошной квадрат из символов *, сторона которого определяется целым параметром **side**. Например, если **side** равно **4**, функция выведет следующее изображение:

```
****
****
****
****
```

5.20. Измените функцию, созданную в упражнении 5.19 таким образом, чтобы она формировала квадрат из любых символов, содержащихся в символьном параметре **fillCharacter**. Например, если **side** равно 5, а символ равен «#», то функция выведет на экран следующее изображение:

```
#####
#####
#####
#####
#####
```

5.21. Используйте методику, аналогичную развитой в упражнениях 5.19 и 5.20 для создания программы, которая рисует разнообразные фигуры.

5.22. Напишите фрагменты программ, которые выполняют следующие действия:

a) Вычисляет целую часть от деления целого числа **a** на целое число **b**.

b) Вычисляет целый остаток от деления целого числа **a** на целое число **b**.

c) Используйте части программы, разработанные в пунктах a) и b), для создания функции, которая вводит целое число между **1** и **32767** и печатает его как ряд цифр, каждая пара которых отделяется двумя пробелами. Например, целое число **4562** должно быть напечатано как

```
4  5  6  2
```

5.23. Напишите функцию, которая получает время в качестве трех целых аргументов (часы, минуты и секунды) и возвращает число секунд с момента, когда часы «пробили 12». Используйте эту функцию для вычисления промежутка времени в секундах между двумя моментами, которые лежат внутри одного и того же 12-ти часового круга.

5.24. Реализуйте следующие целые функции:

a) Функция **celsius** возвращает эквивалент по шкале Цельсия температуры, заданной по шкале Фаренгейта.

b) Функция **fahrenheit** возвращает эквивалент по шкале Фаренгейта температуры, заданной по шкале Цельсия.

c) Используйте эти функции для создания программы, которая печатает таблицы, показывающие температурные эквиваленты шкалы Фаренгейта для всех значений температуры по шкале Цельсия в диапазоне от 0 до 100 градусов, и эквиваленты шкалы Цельсия для всех температур шкалы Фаренгейта в диапазоне от 32 до 212 градусов. Напечатайте аккуратную таблицу в формате, который минимизирует число выводимых строк при сохранении ясности таблицы.

5.25. Напишите функцию, которая возвращает наименьшее из трех чисел с плавающей точкой.

5.26. Целое число называют *совершенным числом*, если сумма его делителей, включая 1 (но не само число), равна этому числу. Например, 6 является совершенным числом, поскольку 6 = 1 + 2 + 3. Напишите функцию **perfect**, которая определяет, является ли ее параметр совершенным числом. Используйте эту функцию в программе, которая находит и печатает все совершенные числа в диапазоне от 1 до 1000. Напечатайте все делители для каждого совершенного числа, чтобы убедиться, что число действительно является совершенным. Бросьте вызов своему компьютеру, попробовав проверить числа, намного большие 1000.

5.27. Целое число называют *простым*, если оно делится только на 1 и на само себя. Например, 2, 3, 5 и 7 являются простыми, а 4, 6, 8 и 9 — нет.

a) Напишите функцию, которая определяет, является ли число простым.

b) Используйте эту функцию в программе, которая находит и печатает все простые числа в диапазоне от 1 до 10000. Сколько чисел из этих 10000 вы действительно должны проверить, чтобы быть уверенным в том, что вы нашли все простые числа?

c) Первоначально вы могли подумать, что верхним пределом для делителей, с которыми нужно проверить простое число, является **n/2**, но в действительности вам нужно дойти только до значения квадратного корня из **n**. Почему? Перепишите программу и запустите ее двумя способами. Оцените увеличение производительности.

5.28. Напишите функцию, которая получает целое значение и возвращает число с обращенным порядком цифр. Например, для числа 7631 функция должна вернуть значение 1367.

5.29. *Наибольший общий делитель* (НОД) двух целых чисел является самым большим целым числом, на которое делится каждое из двух чисел. Напишите функцию **gcd**, которая возвращает наибольший общий делитель двух целых чисел.

5.30. Напишите функцию **gualityPoints**, которая вводит средний бал студента и возвращает 4, если средний бал студента составляет 90-100, 3, если 80-89, 2, если 70-79, 1, если 60-69, и 0, если средний бал ниже чем 60.

5.31. Напишите программу, которая моделирует подбрасывание монеты. Для каждого подбрасывания монеты программа должна печатать слова **Heads** или **Tails** (орел или решка). Пусть программа подбросит

монету 100 раз и сосчитает число выпадений для каждой стороны монеты. Напечатайте результаты. Программа должна вызывать отдельную функцию **flip**, которая не получает никаких аргументов и возвращает **0** для решки и **1** для орла. Примечание: если программа моделирует подбрасывание монеты реалистично, то каждая сторона монеты должна выпасть примерно равное число раз (т.е. примерно 50 орлов и 50 решек при 100 подбрасываниях).

5.32. Компьютеры играют все возрастающую роль в образовании. Напишите программу, которая поможет ученику начальной школы выучить таблицу умножения. Используйте функцию **rand** для генерации двух положительных одноразрядных целых чисел. Программа должна выводить вопрос вроде:

```
How much is 6 times 7? (Сколько будет шестью семь?)
```

Школьник должен напечатать ответ. Ваша программа проверяет ответ. Если он правильный, напечатайте фразу **«Очень хорошо!»**. После этого задавайте следующий вопрос по умножению. Если ответ неправильный, напечатайте **«Неверно. Пожалуйста попробуйте снова.»** и разрешите школьнику отвечать на вопрос до тех пор, пока он не даст правильный ответ.

5.33. Использование компьютеров в образовании относится к области исследований, называющейся *машинным обучением*. Одной из проблем, разрабатываемой в области машинного обучения, является проблема усталости ученика. Усталость может быть уменьшена путем внесения разнообразия в диалог с компьютером, что помогает удержать внимание ученика. Измените программу упражнения 5.32 так, что она будет печатать различные комментарии для каждого правильного ответа и каждого неправильного ответа. Например, следующие:

Реакция на правильный ответ

```
Very good! (Очень хорошо!)
Excellent! (Превосходно!)
Nice work! (Отличная работа!)
Keep up the good work! (Продолжайте в том же духе!)
```

Реакция на неправильный ответ

```
No. Please try again. (Нет. Пожалуйста пробуйте снова.)
Wrong. Try once more. (Неправильно. Попытайтесь еще раз.)
Don't give up! (Не сдавайтесь!)
No. Keep trying. (Нет. Продолжайте пробовать.)
```

Используйте генератор случайных чисел в диапазоне от 1 до 4 для выбора одного из комментариев к каждому ответу. Используйте для вывода фраз структуру **switch** совместно с оператором **printf**.

5.34. Более сложные системы машинного обучения контролируют эффективность работы ученика в течение определенного периода времени. Решение начать новый раздел часто основано на успехах ученика по предыдущим разделам. Измените программу упражнения 5.33 так, чтобы считать число правильных и неправильных ответов, вводимых учеником. После того как ученик введет 10 ответов, ваша программа должна вычислить процент правильных ответов. Если про-

цент ниже чем 75, ваша программа должна напечатать фразу типа **«Пожалуйста, попросите вашего учителя о дополнительной помощи»** и завершить работу.

5.35. Напишите программу С, которая играет в игру «угадай число» следующим образом: ваша программа «задумывает» число, которое нужно угадать, выбирая целое число в диапазоне от 1 до 1000 случайным образом. После этого программа печатает:

```
У меня есть число между 1 и 1000.
Вы можете отгадать мое число?
Пожалуйста, напечатайте ваше первое предположение.
```

После этого игрок вводит первое число. Программа отвечает одной из следующих фраз:

```
1.Превосходно! Вы угадали число!
Не желаете попробовать еще раз?
2.Слишком маленькое. Попробуйте еще раз.
3.Слишком большое. Попробуйте еще раз.
```

Если предположение игрока неправильно, ваша программа должна выполнять цикл, пока игрок не введет правильное число. Ваша программа должна продолжать печатать сообщения игроку **«Слишком большое»** или **«Слишком маленькое»**, чтобы помочь ему попасть в правильный ответ. Примечание: методика поиска, использованная в этой задаче, называется двоичным поиском. В следующей задаче мы поговорим об этом более подробно.

5.36. Измените программу упражнения 5.35 таким образом, чтобы считать число предположений, которое делает игрок. Если это число — 10 или меньше, то напечатайте фразу **«Либо вы знаете секрет, либо вам повезло!»**. Если игрок угадывает число с 10-ти попыток, то напечатайте фразу **«Вы знаете секрет!»**. Если игрок делает более 10-ти предположений, то напечатайте фразу **«У вас есть возможность угадывать быстрее!»**. Почему этот процесс должен длиться не больше 10-ти предположений? С каждым «хорошим предположением» игрок может устранить половину чисел. Покажите, почему любое число от 1 до 1000 можно угадать за 10 или за меньшее число попыток.

5.37. Напишите рекурсивную функцию **power (base, exponent)**, которая возвращает значение

$base^{exponent}$

Например, **power(3, 4) = 3 * 3 * 3 * 3**. Предположим, что **exponent** является целым, большим или равным 1. Подсказка: шаг рекурсии мог бы использовать соотношение

$base^{exponent} = base * base^{exponent-1}$

а завершающим условием будет случай, когда значение **exponent** станет равным 1, поскольку

$base^{1} = base$

5.38. Последовательность чисел Фибоначчи

0, 1, 1, 2, 3, 5, 8, 13, 21, ...

начинается с числа 0 и 1 и имеет то свойство, что каждое последующее число является суммой двух предыдущих. a) Напишите нерекурсивную функцию **fibonacci(n)**, которая вычисляет n-ое число Фибоначчи. b) Определите наибольшее число Фибоначчи, которое можно найти на вашей системе. Для этого измените программу части a) таким образом, чтобы для вычисления и возврата чисел Фибоначчи использовать тип **double** вместо **int**. Разрешите программе зациклиться, пока система не выдаст ошибку из-за чрезмерно большого значения.

5.39. *(Ханойская башня)* Каждый подающий надежды ученый должен познакомиться с некоторыми классическими задачами, а Ханойская башня (см. рис. 5.18) — это одна из наиболее известных классических задач. Легенда гласит, что в храме на Дальнем Востоке священники пытаются переместить башню из дисков с одной стойки на другую. Начальная башня имела 64 диска, собранных на одной стойке и упорядоченных от нижней части до верхней по уменьшению размера. Священники пытаются переместить башню с этой стойки на вторую при следующих ограничениях: за один раз перемещается только один диск и ни при каких условиях диск большего размера не может находиться выше диска меньшего размера. Третья стойка может использоваться для временного хранения дисков. Согласно легенде миру придет конец, когда священники выполнят свою задачу. Так что у нас не так уж много оснований, чтобы попытаться им помочь.

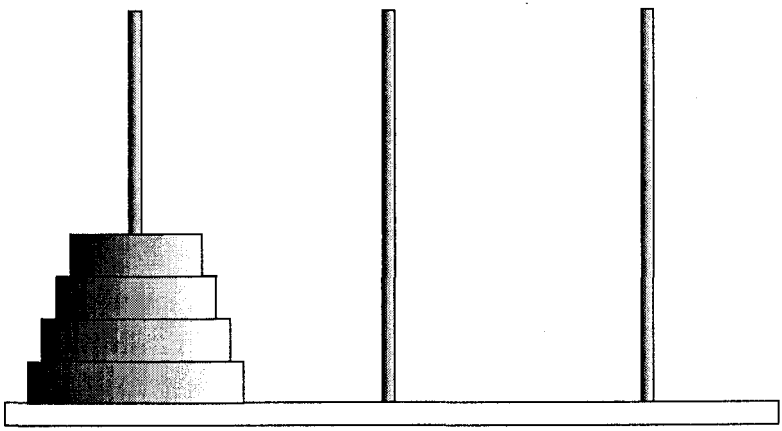


Рис. 5.18. Ханойская башня для случая четырех дисков

Давайте примем, что священники пытаются перемещать диски со стойки 1 на стойку 3. Мы хотим разработать алгоритм, который будет печатать точную последовательность перемещения дисков.

Если мы попробуем подойти к этой задаче со стандартными методами, мы быстро и безнадежно завязнем в перемещениях дисков. Вместо этого, если мы приступим к проблеме, помня о рекурсии, то задача незамедлительно станет послушной. Перемещение n дисков может рассматриваться с точки зрения перемещения только n - 1 диска (а следовательно, это имеет отношение к рекурсии) следующим образом:

1. Переместить n - 1 диск со стойки 1 на стойку 2, используя стойку 3 как временное хранилище.

2. Переместить последний диск (наибольший) со стойки 1 на стойку 3.

3. Переместить n - 1 диск со стойки 2 на стойку 3, используя стойку 1 как временное хранилище.

Этот процесс заканчивается, когда последняя задача потребует перемещения n = 1 диска, т.е. достигается основной случай. Это выполняется обычным перемещением диска без необходимости использовать временное хранилище.

Напишите программу для решения задачи Ханойской башни. Используйте рекурсивную функцию с четырьмя параметрами:

1. Число дисков, которое нужно переместить
2. Стойка, на которой первоначально находятся эти диски
3. Стойка, на которую должна переместиться башня дисков
4. Стойка, которую нужно использовать как временное хранилище

Ваша программа должна выводить точные команды, которые она будет выполнять при перемещении дисков с начальной стойки на конечную стойку. Например, чтобы переместить башню из трех дисков со стойки 1 на стойку 3, ваша программа должна напечатать следующий ряд шагов:

1 → 3 (Это означает перемещение диска со стойки 1 на стойку 3)
1 → 2
3 → 2
1 → 3
2 → 1
2 → 3
1 → 3

5.40. Любая программа, которая может быть реализована рекурсивно, может быть реализована итеративно, хотя иногда со значительно большими трудностями и со значительно меньшей ясностью. Попробуйте создать итеративную версию программы для Ханойской башни. Если вы сделали это успешно, сравните вашу итеративную версию с рекурсивной версией, которую вы разрабатывали в упражнении 5.39. Исследуйте вопросы эффективности, ясности, и ваши возможности продемонстрировать правильность программ.

5.41. *(Визуализация рекурсии)* Интересно понаблюдать рекурсию «в действии». Измените функцию факториала рис. 5.14 так, чтобы она печатала значения своей локальной переменной и параметр рекурсивного вызова. Для каждого рекурсивного вызова выведите значения отдельной строкой и добавьте отступы. Сделайте все возможное, чтобы результат вывода выглядел ясно, интересно и осмысленно. Ваша цель состоит в том, чтобы разработать такой выходной формат данных, который помогает лучше понять рекурсию. Вы можете добавить такие же методы отображения информации к большинству других примеров рекурсии, приведенных в книге.

5.42. Наибольший общий делитель целых чисел **x** и **y** является наибольшим целым числом, на которое делятся и **x** и **y**. Напишите рекурсивную функцию **gcd**, которая возвращает наибольший общий делитель **x** и **y**. Рекурсивное определение функции **gcd** выглядит следующим образом: если **y** равен **0**, то **gcd(x, y)** есть **x**, иначе **gcd(x, y)** есть **gcd(y, x % y)**, где % — операция деления по модулю.

5.43. Может ли **main** быть вызвана рекурсивно? Напишите программу, содержащую функцию **main**. Включите статическую локальную переменную **count**, инициализируемую значением 1. Выполняйте постдекремент и печатайте значение **count** каждый раз, когда вызывается **main**. Запустите программу. Что происходит?

5.44. В упражнениях с 5.32 по 5.34 была разработана программа машинного обучения, которая учит школьников начальных классов умножению. Это упражнение предлагает ввести в программу расширения.

a) Измените программу, чтобы разрешить пользователю вводить уровень сложности задач. Уровень сложности 1 означает, что в задаче используются только одноразрядные числа, уровень сложности 2 означает использование чисел с двумя десятичными разрядами и т.д.

b) Измените программу таким образом, чтобы позволить пользователю выбирать тип арифметических задач, который он хочет исследовать. Опция 1 означает, что будут использованы только задачи на сложение, 2 — только задачи на вычитание, 3 — только задачи на умножение, 4 — только задачи на деление и 5 — смешанное использование всех арифметических действий.

5.45. Напишите функцию **distance**, которая вычисляет расстояние между двумя точками с координатами **(x1, y1)** и **(x2, y2)**. Все числа и возвращаемые значения должны иметь тип **float**.

5.46. Что делает следующая программа?

```
main()
{
   int c;

   if ((c = getchar()) != EOF) {
      main();
      printf("%c", c);
   }

   return 0;
}
```

5.47. Что делает следующая программа?

```
int mystery(int, int);

main()
{
   int x, y;

   printf("Enter two integers: ");
```

```
   scanf("%d%d", &x, &y);
   printf("The result is  %d\n", mystery(x, y));
   return 0;
}

/* Параметр b должен быть положительным
целым, чтобы предотвратить бесконечную рекурсию */
int mystery(int a, int b)
{
   if (b == 1)
      return a;
   else
      return a + mystery(a, b - 1);
}
```

5.48. После того, как вы определили, что делает программа упражнения 5.47, измените программу так, чтобы она правильно работала после удаления ограничения на второй аргумент, который теперь может быть не только положительным.

5.49. Напишите программу, которая проверяет как можно больше функций математической библиотеки (рис. 5.2). Поупражняйтесь с каждой из этих функций, распечатав таблицы возвращаемых значений для различных значений параметров.

5.50. Найдите ошибку в каждом из следующих фрагментов программ и объясните, как ее исправить:

a)
```
float cube(float); /* прототип функции */
...
cube(float number) /* определение функции */
{
   return number * number * numbers
}
```

b)
```
register auto int x = 7;
```

c)
```
int randomNumber = srand();
```

d)
```
float y = 123.45678;
int x;

x = y;
printf("%f\n", (float) x);
```

e)
```
double square(double number)
{
   douhle number;

   return number * number;
}
```

f)
```
int sum(int n)
{
   if (n == 0)
      return 0;
   else
      return n + sum(n);
}
```

5.51. Измените программу игры в крепс на рис. 5.10, чтобы игрок мог делать ставки. Реализуйте как функцию ту часть программы, которая моделирует один розыгрыш. Инициализируйте переменную **bankBalance** значением 1000 долларов. Попросите игрока ввести ставку. Используйте цикл **while** для проверки того, что ставка меньше или равна значению **bankBalance** и, если это не так, запрашивайте пользователя до тех пор, пока он не введет корректную ставку. После ввода корректной ставки программа должна провести один розыгрыш. Если игрок выиграл, увеличьте значение **bankBalance** на величину ставки и напечатайте это новое значение. Если игрок проиграл, уменьшите значение **bankBalance** на величину ставки, напечатайте его, проверьте, не равно ли значение **bankBalance** нулю. Если деньги кончились, напечатайте соответствующее сообщение, например, **«Извините. Вы проиграли!»**. По ходу игры выводите на печать сообщения, создающие впечатление обычной «болтовни». Например, **«О, идете ва-банк, да?»**, **«Ну-ну, рискните!»**, **«Не пора ли вам обменять фишки на наличные?»**.

Массивы



Цели

- Познакомиться со структурой данных, называемой массивом.
- Изучить применение массивов для хранения, сортировки данных и поиска в списках и таблицах значений.
- Научиться объявлять массив, инициализировать его и ссылаться на отдельные элементы массива.
- Научиться передавать массив в функцию.
- Изучить основные методики сортировки.
- Научиться объявлять массивы с несколькими индексами и работать с ними.

Содержание

6.1. Введение
6.2. Массивы
6.3. Объявление массивов
6.4. Примеры использования массивов
6.5. Передача массивов в функции
6.6. Сортировка массивов
6.7. Пример: определение среднего, медианы и наиболее вероятного значения с использованием массивов
6.8. Поиск в массивах
6.9. Многомерные массивы

Резюме • Распространенные ошибки программирования • Хороший стиль программирования • Советы по повышению эффективности • Общие методические замечания • Упражнения для самоконтроля • Ответы к упражнениям для самоконтроля • Упражнения • Упражнения на рекурсию

6.1. Введение

Эта глава служит введением в важную тему о структурах данных. *Массивы* представляют собой структуры данных, состоящие из логически связанных элементов данных одного и того же типа. В главе 10 мы обсудим понятие **struct** (структуры) языка C — структуру данных, состоящую из логически связанных элементов данных, возможно, различных типов. Массивы и структуры являются «статическими» объектами в том смысле, что их размер остается одним и тем же в процессе выполнения программы (конечно, они могут принадлежать к автоматическому классу памяти и, таким образом, создаваться и уничтожаться всякий раз при входе и выходе из блоков, в которых они определяются). В главе 12 мы представим динамические структуры данных, такие как списки, очереди, стеки и деревья, которые могут увеличивать и уменьшать свой размер в процессе выполнения программы.

6.2. Массивы

Массив является группой ячеек памяти, логически связанных в том отношении, что все они имеют одно и то же имя и один и тот же тип. Для обращения к конкретной области, или элементу массива, мы указываем имя массива и *номер позиции* этого элемента в массиве.

На рис. 6.1 показан целочисленный массив **c**. Этот массив содержит двенадцать *элементов*. На каждый из этих элементов можно ссылаться, задавая имя массива и номер позиции конкретного элемента в квадратных скобках (**[]**). Первый элемент в любом массиве имеет *нулевой* порядковый номер. Таким образом, первый элемент массива **c** обозначается как **c[0]**, второй элемент как **c[1]**, седьмой элемент как **c[6]** и, в общем случае, i-й элемент массива **c** обозначается как **c[i-1]**. Для имен массивов справедливы те же соглашения, что и для имен других переменных.

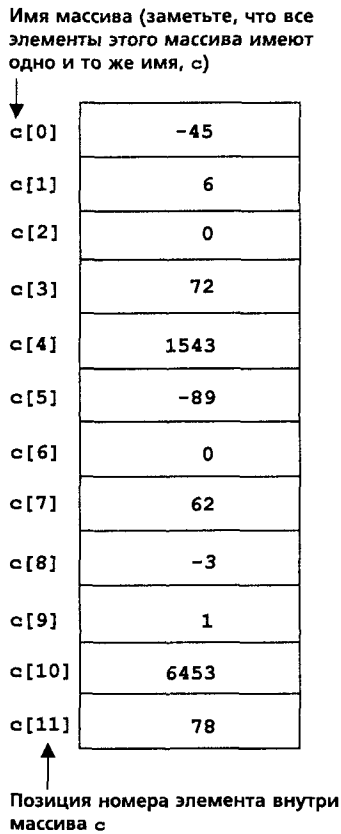


Рис. 6.1. Массив из 12 элементов

Номер позиции элемента, содержащийся внутри квадратных скобок, более формально называют *индексом*. Индекс должен быть целым числом или целочисленным выражением. Если в программе в качестве индекса используется выражение, то для определения индекса происходит оценка этого выражения. Например, если **a = 5** и **b = 6**, то оператор

```
c[a + b] += 2;
```

прибавляет 2 к элементу массива **c[11]**. Обратите внимание, что снабженное индексом имя массива является lvalue — оно может стоять в левой части операции присваивания.

Давайте рассмотрим массив **c**, представленный на рис. 6.1, более подробно. *Имя* этого массива **c**. Его двенадцать элементов обозначаются как **c[0]**, **c[1]**, **c[2]**, ..., **c[11]**. *Значение* **c[0]** равно **-45**, значение **c[1]** равно **6**, значение **c[2]** равно **0**, значение **c[7]** равно **62** и значение **c[11]** равно **78**. Для того, чтобы вывести сумму значений, содержащихся в первых трех элементах массива **c**, мы можем написать

```
printf("%d", c[0] + c[1] + c[2]);
```

Для того, чтобы разделить значение седьмого элемента массива **c** на **2** и присвоить результат переменной **x**, можно написать

```
x = c[6] / 2;
```

Распространенная ошибка программирования 6.1

Важно понимать различие между «седьмым элементом массива» и «элементом массива номер семь». Поскольку индексы массива начинаются с 0, «седьмой элемент массива» имеет индекс 6, в то время как «элемент массива номер семь» имеет индекс 7 и фактически является восьмым элементом массива. Это источник так называемых ошибок смещения индекса.

Скобки, в которые заключается индекс массива, на самом деле рассматриваются в качестве операции языка C. Они имеют тот же самый приоритет, что и круглые скобки. В таблице на рис. 6.2 показаны приоритет и ассоциативность операций, уже представленных в книге. Операции показаны сверху вниз в порядке уменьшения их приоритета.

Операции	Ассоциативность	Тип
`()  []`	*слева направо*	*наивысший*
`++  --  !  (тип)`	справа налево	унарные
`*  /  %`	слева направо	мультипликативные
`+  -`	слева направо	аддитивные
`<  <=  >  >=`	слева направо	отношения
`==  !=`	*слева направо*	*равенства*
`&&`	слева направо	логическое И
`||`	слева направо	логическое ИЛИ
`?:`	справа налево	условные
`=  +=  -=  *=  /=  %=`	справа налево	присваивания
`,`	слева направо	запятая

Рис. 6.2. Приоритет операций

6.3. Объявление массивов

Массивы занимают определенное место в памяти. Программистом устанавливается тип каждого элемента и число элементов, требуемых для каждого массива, чтобы компьютер мог зарезервировать соответствующий объем памяти. Для указания компьютеру, чтобы он зарезервировал 12 элементов для целочисленного массива **c**, нужно написать

```
int c[12];
```

При помощи одного объявления можно зарезервировать память для нескольких массивов. Чтобы зарезервировать 100 элементов для целочисленного массива **b** и 27 элементов для целочисленного массива **x**, нужно написать:

```
int b[100], x[27];
```

Массивы могут объявляться также и для хранения других типов данных. Например, для символьной строки может использоваться массив типа **char**. Символьные строки и их сходство с массивами рассматриваются в главе 8. Связь между указателями и массивами обсуждается в главе 7.

6.4. Примеры работы с массивами

Программа, представленная на рис. 6.3, использует структуру повторения **for** для инициализации нулями элементов целочисленного массива **n** из десяти элементов и выводит его в табличной форме.

```
/* инициализация массива */
#include <stdio.h>

main()
{
   int n[10], i;

   for (i = 0; i <= 9; i++)
      n[i] = 0;                    /* инициализация массива */

   printf("%s%13s\n", "Element", "Value");
   for (i = 0; i <= 9; i++)        /* вывод элементов массива */
      printf("%7d%13d\n", i, n[i]);

   return 0;
}
```

```
Element         Value
      0             0
      1             0
      2             0
      3             0
      4             0
      5             0
      6             0
      7             0
      8             0
      9             0
```

Рис. 6.3. Инициализация элементов массива нулями

Обратите внимание, что мы решили не помещать пустую строку между первым оператором **printf** и структурой **for** на рис. 6.3, поскольку эти операторы тесно связаны. В данном случае оператор **printf** отображает заголовки для двух столбцов, выводимых в структуре **for**. Программисты часто опускают пустую строку между структурой **for** и тесно связанным с ней оператором **printf**.

Элементы массива также могут инициализироваться при объявлении массива путем помещения вслед за его объявлением знака равенства и списка (заключенного в фигурные скобки) *инициализирующих значений*, разделенных запятыми. Программа, представленная на рис. 6.4, инициализирует целочисленный массив десятью значениями и выводит его в табличной форме.

```
/ * Инициализация массива при объявлении * /
#include <stdio.h>

main()
{
   int i, n[10] = {32, 27, 64, 18, 95, 14, 90, 70, 60, 37};

   printf("%s%13s\n", "Element", "Value");

   for (i = 0; i <= 9; i++)
      printf("%7d%13d\n", i, n[i]);

   return 0;
}
```

```
Element         Value
      0            32
      1            27
      2            64
      3            18
      4            95
      5            14
      6            90
      7            70
      8            60
      9            37
```

Рис. 6.4. Инициализация элементов массива при объявлении

Если инициализирующих значений меньше, чем элементов массива, остающиеся элементы автоматически инициализируются нулями. Например, элементы массива **n** на рис. 6.3 можно было бы инициализировать нулями посредством объявления

```
int n[10] = {0};
```

которое явно инициализирует нулем первый элемент и автоматически инициализирует нулями оставшиеся девять элементов, поскольку инициализирующих значений меньше, чем элементов в массиве. Важно помнить, что массивы не инициализируются нулями автоматически. Программист должен инициализировать по крайней мере первый элемент, чтобы оставшиеся элементы были автоматически заполнены нулями. Этот метод инициализации элементов массива нулями реализуется во время компиляции. Метод, используемый на рис. 6.3, может применяться многократно в процессе выполнения программы.

Распространенная ошибка программирования 6.2

Отсутствие инициализации массива, элементы которого должны быть инициализированы.

Следующее объявление массива

```
int n[5] = {32, 27, 64, 18, 95, 14};
```

вызвало бы синтаксическую ошибку, поскольку имеется 6 инициализирующих значений и только 5 элементов массива.

Распространенная ошибка программирования 6.3

Указание в списке инициализации большего числа значений, чем количество элементов в массиве, является синтаксической ошибкой.

Если размер массива не включается в его объявление со списком инициализирующих значений, то число элементов в массиве будет равно числу элементов в списке инициализации. Например, объявлением

```
int n[] = {1, 2, 3, 4, 5};
```

был бы создан массив из пяти элементов.

Программа на рис. 6.5 инициализирует элементы десятиэлементного массива **s** значениями **2**, **4**, **6**, ..., **20** и выводит массив в табличной форме. Эти значения генерируются путем умножения счетчика цикла на **2** и прибавления **2**.

В этой программе вводится директива препроцессора **#define**. Строка

```
#define SIZE 10
```

определяет *символическую константу* **SIZE**, значение которой равно **10**. Символическая константа является идентификатором, вместо которого препроцессор С до компиляции программы подставляет *заменяющий текст*. Во время препроцессорной обработки программы на рис. 6.5 все вхождения символической константы **SIZE** заменяются на текст **«10»**. Использование символических констант для задания размеров массива делает программу *масштабируемой*. На рис. 6.5 с помощью первого цикла **for** можно было бы заполнить массив из 1000 элементов, просто изменив значение **SIZE** в директиве **#define** с **10** на **1000**. Если бы символическая константа **SIZE** не использовалась, то, чтобы масштабировать программу для обработки 1000 элементов массива, нам пришлось бы изменить программу в трех различных местах. По мере увеличения размера кода эта методика становится еще более полезной, способствуя написанию более ясных программ.

Распространенная ошибка программирования 6.4

Завершение директив препроцессора **#define** или **#include** точкой с запятой. Не забывайте, что директивы препроцессора не являются операторами языка С.

Если предыдущая директива препроцессора **#define** будет оканчиваться точкой с запятой, то все вхождения символической константы **SIZE** в программе будут заменены препроцессором на текст **«10;»**. Это может привести к синтаксическим ошибкам времени компиляции или логическим ошибкам во время выполнения. Не забывайте, что препроцессор не является компилятором С, это всего лишь текстовый манипулятор.

```
/* Инициализирует элементы массива s
   четными целыми числами от 2 до 20 */
#include <stdio.h>
#define SIZE 10

main()
{
   int s[SIZE], j;

   for (j = 0; j <= SIZE - 1; j++) /* устанавливает значения */
      s[j] = 2 + 2 * j;

   printf("%s%13s\n", "Element", "Value");

   for (j = 0; j <= SIZE - 1; j++) /* выводит значения */
      printf("%7d%13d\n", j, s[j]);

   return 0;
}
```

```
Element         Value
      0             2
      1             4
      2             6
      3             8
      4            10
      5            12
      6            14
      7            16
      8            18
      9            20
```

Рис. 6.5. Генерация значений для размещения в элементах массива

Распространенная ошибка программирования 6.5

Присвоение значения символической константе в исполняемом операторе является синтаксической ошибкой. Символическая константа — не переменная. Компилятор не резервирует для нее места в памяти, как в случае переменных, содержащих значения во время выполнения программы.

Общее методическое замечание 6.1

Определение размера всех массивов с помощью символических констант повышает общность программ (делает их масштабируемыми).

Хороший стиль программирования 6.1

Используйте для имен символических констант только символы верхнего регистра. Это выделяет эти константы в тексте программы и напоминает программисту, что символические константы не являются переменными.

Программа на рис. 6.6 суммирует значения, содержащиеся в целочисленном массиве a из двенадцати элементов. Оператор в теле цикла for вычисляет итоговую сумму.

```
/ * Вычисляет сумму элементов массива * /
#include <stdio.h>
#define SIZE 12

main()
{
   int a[SIZE] = {1, 3, 5, 4, 7, 2, 99, 16, 45, 67, 89, 45},
       i, total = 0;

   for (i = 0; 1 <= SIZE - 1; i++)
      total += a[i];

   printf ("Total of array element values is %d\n", total);
   return 0;
}
```

```
Total of array element values is 383
```

Рис. 6.6. Вычисление суммы элементов массива

В нашем следующем примере массивы используются для подведения итогов опроса. Рассмотрим постановку задачи.

Сорок студентов попросили оценить качество питания в студенческой столовой по шкале от 1 до 10 (1 означает ужасное качество, а 10 означает превосходное). Поместите сорок ответов в целочисленный массив и определите итоговый результат опроса.

Это типичное применение массива (см. рис. 6.7). Мы хотим просуммировать количество ответов каждого типа (т.е. от 1 до 10). Массив **responses** является 40-элементным массивом ответов студентов. Мы используем массив из одиннадцати элементов **frequency** для подсчета числа появлений каждого ответа. Мы игнорируем первый элемент **frequency[0]**, поскольку более логично, чтобы ответ 1 увеличивал **frequency[1]**, а **не frequency[0]**. Это позволяет нам использовать каждый тип ответа непосредственно в качестве индекса в массиве **frequency**.

Хороший стиль программирования 6.2

Стремитесь к ясности программы. Иногда имеет смысл отказаться от наиболее эффективного использования памяти или процессорного времени в пользу написания более понятного кода.

Совет по повышению эффективности 6.1

Иногда соображения эффективности намного перевешивают соображения понятности.

В первом цикле **for** из массива responses по одному выбираются ответы и в массиве **frequency** увеличивается один из десяти счетчиков (от **frequency[1]** до **frequency[10]**). Ключевым оператором цикла является

```
++frequency[responses[answer]];
```

Этот оператор увеличивает соответствующий счетчик **frequency** в зависимости от значения **responses[answer]**. Например, когда переменная-счетчик **answer** равна **0**, **responses[answer]** равно **1**, так что **++frequency[responses[answer]];** на самом деле интерпретируется как

```
++frequency[1];
```

что увеличивает элемент массива номер один.

```
/ * Программа опроса студентов * /
#include <stdio.h>
#define RESPONSE_SIZE 40
#define FREQUENCY_SIZE 11

main()
{
   int answer, rating;
   int responses[RESPONSE_SI2E] = {1, 2, 6, 4, 8, 5, 9, 7, 8,
       10, 1, 6, 3, 8, 6, 10, 3, 8, 2, 7, 6, 5, 7, 6, 8, 6, 7,
       5, 6, 6, 5, 6, 7, 5, 6, 4, 8, 6, 8, 10};
   int frequency[FREQUENCY_SIZE] = {0};

   for (answer = 0; answer <= RESPONSE_SIZE - 1; answer++)
      ++frequency[responses[answer]];

   printf("%s%17s\n", "Rating", "Frequency");

   for (rating = 1; rating <= FREQUENCY_SIZE - 1; rating++)
      printf("%6d%17d\n", rating, frequency[rating]);

   return 0;
}
```

```
Rating          Frequency
     1                  2
     2                  2
     3                  2
     4                  2
     5                  5
     6                 11
     7                  5
     8                  7
     9                  1
    10                  3
```

Рис. 6.7. Простая программа анализа опроса студентов

Когда переменная **answer** равна **1**, **responses[answer]** равно **2**, так что **++frequency[responses[answer]];** интерпретируется как

```
++frequency[2];
```

что увеличивает элемент массива номер два. Когда переменная **answer** равна **2**, **responses[answer]** равно **6**, так что **++frequency[responses[answer]];** интерпретируется как

```
++frequency[6];
```

что увеличивает элемент массива номер шесть, и так далее. Обратите внимание, что вне зависимости от числа ответов, обработанных во время обследования, для суммирования результатов требуется массив, содержащий только одиннадцать элементов (мы игнорируем элемент с номером нуль). Если бы данные содержали недопустимые значения, как, например **13**, программа попыталась бы прибавить **1** к **frequency[13]**. Это было бы выходом за пределы массива.

В языке С отсутствует проверка на выход за пределы массива, предотвращающая ссылку на несуществующий элемент. Таким образом, выполняющаяся программа может выйти за пределы массива без предупреждения. Программист должен гарантировать, что все ссылки на массив не выходят за его пределы.

Распространенная ошибка программирования 6.6

Ссылка на элемент за пределами массива.

Хороший стиль программирования 6.3

При выполнении цикла над элементами массива его индекс никогда не должен быть меньше 0 и должен быть строго меньше общего количества элементов, т.е. не больше, чем (размер массива - 1). Убедитесь, что условие завершения цикла не допускает обращения к элементам вне этого диапазона.

Хороший стиль программирования 6.4

Указывайте в структуре **for** наибольшее значение индекса массива, способствуя тем самым устранению ошибок преждевременного выхода из цикла.

Хороший стиль программирования 6.5

Программы должны проверять правильность всех входных значений, чтобы не допустить воздействия ошибочной информации на вычисления программы.

Совет по повышению эффективности 6.2

Последствия (обычно серьезные) ссылок на элементы за пределами массива являются системно-зависимыми.

В нашем следующем примере (рис. 6.8) происходит считывание чисел из массива и графическое отображение информации в виде столбцовой диаграммы, или гистограммы; выводится каждое число и, кроме того, рядом с ним выводится полоса, состоящая из такого же количества звездочек. Полосы выводятся вложенным циклом **for**. Обратите внимание на использование **printf("\n")** для завершения полосы гистограммы.

В главе 5 мы обещали, что покажем более изящный метод написания программы для бросания кости, приведенной на рис. 5.10. Задача состояла в бросании 6000 раз шестигранной кости для проверки того, на самом ли деле генератор случайных чисел вырабатывает случайные числа. Вариант этой программы с массивом показан на рис. 6.9.

До сих пор мы обсуждали только целочисленные массивы. Однако в массивах могут содержаться данные любого типа. Теперь мы обсудим хранение строк в символьных массивах. Пока единственной возможностью обработки строк, которой мы располагаем, является вывод строки с помощью функции **printf**. В языке С строка типа **"hello"** на самом деле является массивом отдельных символов.

```
/ * Программа печати гистограммы * /
#include <stdio.h>
#define SIZE 10

main()
{
   int n[SIZE] = {19, 3, 15, 7, 11, 9, 13, 5, 17, 1};
   int i, j;

   printf("%s%13s%17s\n", "Element", "Value", "Histogram");

   for (i = 0; i <= SIZE - 1; i++) {
      printf("%7d%13d    ", i, n[i]);

      for (j = 1; j < n[i]; j++)  /* выводит отдельный столбец */
         printf("%c", '*');

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

```
Element         Value           Histogram
      0            19           *******************
      1             3           ***
      2            15           ***************
      3             7           *******
      4            11           ***********
      5             9           *********
      6            13           *************
      7             5           *****
      8            17           *****************
      9             1           *
```

Рис. 6.8. Программа вывода гистограмм

Символьные массивы имеют несколько уникальных особенностей. Символьный массив может быть инициализирован строковым литералом. Например, объявление

```
char string1[] = "first";
```

инициализирует элементы массива **string1** отдельными символами строкового литерала **"first"**. Размер массива **string1** в предыдущем объявлении определяется компилятором исходя из длины строки.

Важно обратить внимание на то, что строка **"first"** содержит пять символов *и* специальный символ окончания строки, называемый *нулевым символом*. Таким образом, массив **string1** фактически содержит шесть элементов. Представлением нулевого символа в виде символьной константы является **'\0'**. Все строки в языке C заканчиваются этим символом. Символьный массив, представляющий строку, всегда следует объявлять достаточно большим для хранения в нем всех символов строки и завершающего нулевого символа.

Символьные массивы могут также инициализироваться отдельными символьными константами в инициализирующем списке. Предыдущее объявление эквивалентно

```
char string1 = {'f', 'i', 'r', 's', 't', '\0'};
```

```
/ * Моделирует бросание шестигранной кости 6000 раз * /
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define SIZE 7

main()
{
   int face, roll, frequency[SIZE] = {0};

   srand(time(NULL));

   for (roll = 1; roll <= 6000; roll++) {
      face = rand() % 6 + 1;

      ++frequency[face];  /* заменяет 20-строчный switch */
   }                      /* на рис. 5.8 */

   printf("%s%17s\n", "Face", "Freduency");

   for (face = 1; face <= SIZE - 1; face++)
      printf("%4d%17d\n", face, frequency[face]);

   return 0;
}
```

```
Face          Frequency
   1               1037
   2                987
   3               1013
   4               1028
   5                952
   6                983
```

Рис. 6.9. Программа бросания кости, использующая массивы вместо оператора switch

Поскольку строка в действительности является символьным массивом, мы можем непосредственно обращаться к отдельным символам строки, пользуясь индексной нотацией. Например, **string1[0]** представляет собой символ **'f'**, а **string1[3]** — **'s'**.

Мы также можем вводить строку в символьный массив непосредственно с клавиатуры, пользуясь для этого функцией **scanf** и спецификацией преобразования **%s**. Например, объявление

```
char string2[20];
```

создает символьный массив, в котором могут храниться строка из 19 символов и завершающий нулевой символ. Оператор

```
scanf("%s", string2);
```

считывает строку с клавиатуры в **string2**. Обратите внимание, что имя массива передается в функцию **scanf** без предшествующего &, необходимого с другими переменными. Символ & обычно используется для того, чтобы снабдить **scanf** информацией о местоположении переменной в памяти, с тем чтобы там можно было сохранить в ней значение. В разделе 6.5 мы обсудим передачу массивов функциям. Мы увидим, что имя массива является адресом его начала; поэтому символ & не является необходимым.

Ответственность за то, чтобы в массиве, куда считываются символы, могла быть сохранена любая вводимая пользователем строка, лежит на программисте. Приведенный выше вызов функции **scanf** считывает символы с клавиатуры до тех пор, пока не будет встречен первый пробельный символ — размер массива ее не интересует. Таким образом, функция **scanf** может записывать информацию за пределы массива.

Распространенная ошибка программирования 6.7

Передача функции **scanf** символьного массива, недостаточно большого для хранения вводимой с клавиатуры строки, может привести к потере данных в программе и другим ошибкам времени выполнения.

Символьный массив, представляющий строку, может быть выведен с помощью функции **printf** и спецификатора преобразования **%s**. Массив **string2** выводится при помощи оператора

```
printf("%s\n", string2);
```

Обратите внимание, что **printf**, как и **scanf**, не заботится о размерах символьного массива. Символы строки выводятся до тех пор, пока не будет встречен завершающий нулевой символ.

Рис. 6.10 демонстрирует инициализацию символьного массива строковым литералом, чтение строки в символьный массив, вывод символьного массива в виде строки и доступ к отдельным символам строки.

```
/ * Интерпретация символьных массивов в виде строк * /
#include <stdio.h>

main()
{
   char string1[20], string2[] = "string literal";
   int i;

   printf("Enter a string: ");
   scanf("%s", string1);
   printf("string1 is: %s\nstring2: is %s\n"
          "string1 with spaces between characters is:\n",
          string1, string2);

   for (i = 0; string1[i] != '\0'; i++)
      printf("%c ", string1[i]);

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

```
Enter a string: Hello there
string1 is: Hello
string2 is: String literal
string1 with spaces between characters is:
H e l l o
```

Рис. 6.10. Интерпретация символьных массивов в качестве строк

На рис. 6.10 структура **for** используется для прохода в цикле по массиву **string1** и вывода отдельных символов, разделенных пробелами, при помощи

спецификации преобразования %c. Условие в структуре for, **string1[i] != '\0'**, остается истинным, пока в строке не будет встречен завершающий нулевой символ.

В главе 5 обсуждался спецификатор класса памяти **static**. Статическая локальная переменная в определении функции существует на всем протяжении выполнения программы, но видима только в теле функции. Мы можем применять **static** к объявлению локального массива, при этом во время работы программы массив не будет создаваться и инициализироваться всякий раз при вызове функции и уничтожаться при выходе из нее. Это уменьшает время выполнения программ, особенно программ с частым вызовом функций, содержащих большие массивы.

Совет по повышению эффективности 6.3

В функциях, содержащих локальные массивы и часто входящих и выходящих из области действия, делайте массивы статическими, чтобы они не создавались заново при каждом вызове функции.

Массивы, объявленные статическими, автоматически инициализируются один раз во время компиляции. Если статический массив явно не инициализирован программистом, компилятор инициализирует его нулями.

На рис. 6.11 показаны функция **staticArrayInit** с локальным массивом, объявленным **static**, и функция **automaticArrayInit** с автоматическим локальным массивом. Функция **staticArrayInit** вызывается дважды. Статический локальный массив, содержащийся в функции, инициализируется компилятором нулями. Функция выводит массив, прибавляет 5 к каждому его элементу и снова выводит массив. При втором вызове функции статический массив содержит значения, сохранившиеся со времени ее первого вызова. Функция **automaticArrayInit** также вызывается дважды. Элементы автоматического локального массива, содержащегося в функции, инициализируются значениями 1, 2 и 3. Функция выводит массив, прибавляет 5 к каждому его элементу и снова выводит этот массив. При втором вызове функции элементы массива снова инициализируются как 1, 2 и 3, поскольку массив имеет автоматический период хранения.

```
/ * Статические массивы инициализируются нулями * /
#include <stdio.h>

void staticArrayInit(void);
void automaticArrayInit(void);

main()
{
   printf("First call to each function:\n");
   staticArrayInit();
   automaticArrayInit();
   printf(\n\nSecond call to each function:\n");
   staticArrayInit();
   automaticArrayInit();
   return 0;
}
```

Рис. 6.11. Статические массивы автоматически инициализируются нулями, если они явно не инициализированы программистом (часть 1 из 2)

```
/* функция, иллюстрирующая статический локальный массив */
void staticArrayInit(void)
{
   static int a[3];
   int i;

   printf("\nValues on entering staticArrayInit:\n");

   for (i = 0; i <= 2; i++)
      printf("array1[%d] = %d  ", i, a[i]);

   printf["\nValues on exiting staticArrayInit:\n");

   for (i = 0; i <= 2; I++)
      printf("array1[%d] = %d  ", i, a[i] += 5);
}

/* функция, иллюстрирующая автоматический локальный массив */
void automaticArrayInit(void)
{
   int a[3] = {1, 2, 3};
   int i;

   printf("\n\nValues on entering automaticArrayInit:\n");

   for (i = 0; i <= 2; i++)
      printf("array1[%d] = %d  ", i, a[i]);

   printf("\nValues on exiting automaticArrayInit:\n");

   for (i = 0; i <= 2; i++)
      printf("array1[%d] = %d  ", i, a[i] += 5);
}
```

```
First call to each function:

Values on entering staticArrayInit:
array1[0] = 0  array1[1] = 0  array1[2] = 0
Values on exiting staticArrayInit:
array1[0] = 5  array1[1] = 5  array1[2] = 5
Values on entering automaticArrayInit:
array1[0] = 1  array1[1] = 2  array1[2] = 3
Values on exiting automaticArrayInit:
array1[0] = 6  array1[1] = 7  array1[2] = 8

Second call to each function

Values on entering staticArrayInit:
array1[0] = 5  array1[1] = 5  array1[2] = 5
Values on exiting staticArrayInit:
array1[0] = 10 array1[1] = 10 array1[2] = 10

Values on entering automaticArrayInit:
array1[0] = 1  array1[1] = 2  array1[2] = 3
Values on exiting automaticArrayInit:
array1[0] = 6  array1[1] = 7  array1[2] = 8
```

Рис. 6.11. Статические массивы автоматически инициализируются нулями, если они явно не инициализированы программистом (часть 2 из 2)

Распространенная ошибка программирования 6.8

Предположение, что элементы локального массива, объявленного как **static**, инициализируются нулями всякий раз при вызове функции, где этот массив объявлен.

6.5. Передача массивов в функции

Для передачи массива в качестве параметра функции укажите его имя без всяких скобок. Например, если массив **hourlyTemperatures** был объявлен как

```
int hourlyTemeratures[24];
```

оператор вызова функции

```
modifyArray(hourlyTemeratures, 24);
```

передает массив **hourlyTemperatures** и его размер в функцию **modifyArray**. При передаче массива в функцию часто передается и размер массива, чтобы функция смогла обработать определенное количество его элементов.

С автоматически передает массив в функцию путем имитации передачи параметра по ссылке — при этом вызываемые функции могут изменять значения элементов в исходных массивах вызывающих функций. Ведь имя массива фактически является адресом его первого элемента! Поскольку передается начальный адрес массива, вызываемая функция точно знает, где хранится этот массив. Таким образом, когда вызываемая функция изменяет в своем теле элементы массива, она изменяет подлинные элементы массива в их исходных ячейках памяти.

Рис. 6.12 демонстрирует, что имя массива действительно является адресом его первого элемента, выводя значения **array** и **&array[0]** со спецификацией преобразования **%p** — специальной спецификацией для вывода адресов. Спецификация преобразования **%p** обычно выводит адреса виде шестнадцатеричных чисел. Шестнадцатеричные (основание 16) числа состоят из цифр от 0 до 9 и букв от A до F. Они часто используются для сокращенной записи больших целочисленных значений. В приложении Д дано подробное описание взаимоотношений между двоичными (основание 2), восьмеричными (основание 8), десятичными (основание 10; стандартные целые числа) и шестнадцатеричными целыми числами. Выходные данные показывают, что как **array**, так и **&array[0]** имеют одно и то же значение, а именно **FFF0**. Вывод этой программы является системно-зависимым, но адреса всегда будут совпадать.

Совет по повышению эффективности 6.4

Передача массивов посредством имитации передачи по ссылке имеет смысл по соображениям эффективности. Если бы массивы передавались по значению, передавалась бы копия каждого элемента. Для больших часто передаваемых массивов это было бы расточительно по времени и приводило бы к расходу значительного объема памяти для хранения их копий.

Общее методическое замечание 6.2

Возможна передача массива по значению с помощью простого приема, который описывается в главе 10.

```
/ * Имя массива это то же самое, что &array[0] * /
#include <stdio.h>

main()
{
   char array[5];

   printf("   array = %p\n&array[0]  = %p\n",
          array, &array[0]);
   return 0;
}
```

```
    array = FFF0
&array[0] = FFF0
```

Рис. 6.12. Имя массива — это то же самое, что и адрес его первого элемента

Хотя в целом массивы передаются посредством имитации передачи параметра по ссылке, отдельные элементы массива передаются по значению, точно так же, как простые переменные. Такие простые одиночные порции данных называются *скалярами* или *скалярными величинами.* При передаче в функцию элемента массива используйте в качестве параметра его имя и индекс. В главе 7 мы покажем способы имитации передачи параметра по ссылке для скаляров (т.е. отдельных переменных и элементов массива).

Для того, чтобы функция при обращении к ней могла принять массив, в списке параметров этой функции должно быть указано, что ожидается передача массива. Например, заголовок для функции **modifyArray** мог бы быть записан как

```
void modifyArray(int b[], int size)
```

Список параметров показывает, что функция **modifyArray** ожидает приема массива целых чисел в параметре **b** и числа элементов массива в параметре **size**. Размер массива в скобках в заголовке не требуется. При его указании компилятор его игнорирует. Поскольку массивы автоматически передаются путем имитации передачи параметра по ссылке, то, когда вызываемая функция использует имя массива **b**, она на самом деле ссылается на фактически существующий массив в вызывающей функции (массив **hourlyTemperatures** в предыдущем вызове). В главе 7 мы введем другие обозначения для указания того, что функция принимает массив. Мы увидим, что эти обозначения основаны на близкой связи между массивами и указателями в языке C.

Обратите внимание на необычный внешний вид прототипа функции **modifyArray**

```
void modifyArray(int [], int);
```

Этот прототип можно было бы записать в виде

```
void modifyArray(int anyArrayName[], int anyVariableName);
```

однако, как мы узнали в главе 5, компилятор языка C игнорирует имена переменных в прототипах.

Хороший стиль программирования 6.6

Некоторые программисты включают имена переменных в прототипы функций для того, чтобы сделать программы более понятными. Компилятор игнорирует эти имена.

Не забывайте, что прототип сообщает компилятору о количестве параметров и типе каждого из них (в том порядке, в котором эти параметры будут передаваться).

В программе на рис. 6.13 показано различие между передачей целого массива и элемента массива. Сначала программа выводит пять элементов целочисленного массива **a**. Затем массив **a** и его размер передаются в функцию **modifyArray**, в которой каждый элемент массива **a** умножается на 2. После этого в программе **main** происходит повторный вывод массива **a**. Как показывают выводимые данные, функция **modifyArray** действительно изменила элементы массива **a**. Теперь программа выводит значение **a[3]** и передает его в функцию **modifyElement**. Функция **modifyElement** умножает свой параметр на 2 и выводит новое значение. Обратите внимание, что когда элемент **a[3]** повторно выводится в функции **main**, его значение остается прежним, поскольку отдельные элементы массива передаются по значению.

В ваших программах могут возникать ситуации, когда функция не должна изменять элементы массива. Поскольку массивы всегда передаются путем имитации передачи параметра по ссылке, модификация значений в массиве с трудом поддается контролю. Для предотвращения изменения в функции значений элементов массива в языке C предусмотрен специальный модификатор типа **const**. Когда параметру-массиву предшествует **const**, элементы массива становятся константами в теле функции и любая ее попытка изменить элемент массива приводит к ошибке времени компиляции. Это дает возможность программисту исправить программу таким образом, чтобы она не пыталась изменять элементы массива. Хотя модификатор **const** строго определен в стандарте ANSI, в различных системах C способы его поддержки различны.

```
/ * Передача в функцию массива и отдельного элемента массива * /
#include <stdio.h>
#define SIZE 5

void modifyArray(int [], int);     /* выглядит необычно */
void modifyElement(int);

main()
{
   int a[SIZE] = {0, 1, 2, 3, 4};
   int i;

   printf("Effects of passing entire array call "
          "by reference: \n\nThe values of the "
          "original array are:\n");

   for (i = 0; i <= SIZE - 1; i++)
      printf("%3d", a[i]);

   printf("\n");
   modifyArray(a, SIZE); /* массив a передается по ссылке */
   printf("The values of the modified array are: \n");

   for (i = 0; i <= SIZE - 1; i++)
      printf("%3d", a[i]);
```

Рис. 6.13. Передача в функцию массива и его отдельного элемента (часть 1 из 2)

```
   printf("\n\n\nEffects of passing array element call "
          "by value:\n\nThe value of a[3] is %d\n", a[3]);
   modifyElement(a[3]);
   printf("The value of a[3] is %d\n", a[3]);
   return 0;
}

void modifyArray(int b[], int size)
{
   int j;

   for (j = 0; j <= size - 1; j++)
      b[j] *= 2;
}

void modifyElement(int e)
{
   printf("Value in modifyElement is %d\n", e *= 2);
}
```

```
Effects of passing entire array call by reference:

The values of the original array are:
  0  1  2  3  4
The values of the modified array are:
  0  2  4  6  8

Effects of passing array element call by value:

The value of a[3] is 6
Value in modifyElement is 12
The value of a[3] is 6
```

Рис. 6.13. Передача в функцию массива и его отдельного элемента (часть 2 из 2)

Рис. 6.14 демонстрирует описатель **const**. Функция **tryToModifyArray** определена с параметром **const intb[]**, который указывает, что массив **b** является константным и не может быть изменен. Результатом работы программы являются сообщения об ошибках (компилятор Borland C++). Каждая из трех попыток функции изменить элементы массива приводит к ошибке компилятора **«Невозможно изменить константный объект»**. Модификатор **const** снова будет обсуждаться в главе 7.

Общее методическое замечание 6.3

К параметру-массиву в определении функции может применяться модификатор **const**, чтобы не допустить изменения в теле функции исходного массива. Это является еще одним примером принципа наименьших привилегий. Функциям не следует давать возможности изменять массивы, если только это не является совершенно необходимым.

```
/ * Демонстрация модификатора типа const * /
#include <stdio.h>

void tryToModifyArray(conat int []);

main()
{
   int a[] = {10, 20, 30};

   tryToModifyArray(a);
   printf("%d %d %d\n", a[0], a[1], a[2]);
   return 0;
}

void tryToModifyArray(conat int b[])
{
   b[0] /= 2;   /* ошибка */
   b[1] /= 2;   /* ошибка */
   b[2] /= 2;   /* ошибка */
}
```

```
Compiling FIG6_14.C:
Error FIG6_14.C 16: Cannot modify a const object
Error FIG6_14.C 17: Cannot modify a const object
Error FIG6_14.C 18: Cannot modify a const object
Warning FIG6_14.C 19: Parameter 'b' is never used
```

Рис. 6.14. Демонстрация модификатора типа **const**

6.6. Сортировка массивов

Сортировка данных (т.е. расположение данных в некотором специальном порядке, например, по возрастанию или убыванию) является одним из наиболее важных применений компьютера. Банк сортирует все чеки по номерам счетов, чтобы в конце каждого месяца можно было подготовить личные банковские декларации. Телефонные компании сортируют списки своих счетов по фамилиям и, внутри фамилий, по именам, чтобы облегчить поиск телефонных номеров. Практически каждой организации приходится сортировать какие-нибудь данные (и во многих случаях — большие массивы данных). Сортировка данных представляет собой захватывающую проблему, требующую самых интенсивных усилий исследователей в области информатики. В этой главе мы обсудим, возможно, самую простую из известных схем сортировки. В упражнениях и в главе 12 мы исследуем более сложные схемы, которые являются намного более эффективными.

Совет по повышению эффективности 6.5

Часто наиболее простые алгоритмы работают плохо. Их достоинством является простота написания, тестирования и отладки. Однако для достижения максимальной эффективности часто требуются более сложные алгоритмы.

Программа на рис. 6.15 сортирует значения элементов десятиэлементного массива a в порядке их возрастания. Используемая нами методика называется

пузырьковой сортировкой или *сортировкой погружением*, поскольку меньшие значения, подобно воздушным пузырькам в воде, постепенно «всплывают» в верхнюю часть массива, в то время как большие значения опускаются вниз. Метод требует нескольких проходов по массиву. На каждом проходе сравниваются последовательные пары элементов. Если элементы расположены в возрастающем порядке (или если их значения совпадают), мы ничего не меняем. Если элементы пары расположены в убывающем порядке, их значения в массиве меняются местами.

```
/* Программа сортирует значения массива в порядке возрастания */
#include <stdio.h>
#define SIZE 10

main()
{
   int a[SIZE] = {2, 6, 4, 8, 10, 12, 89, 68, 45, 37};
   int i, pass, hold;

   printf("Data items in original order\n");

   for (i = 0; i <= SIZE - 1; i++)
      printf("%4d", a[i]);

   for (pass = 1; pass <= SIZE - 1; pass++)    /* проходы */

      for (i = 0; i <= SIZE - 2; I++)          /* один проход */

         if (a[i] > a[i + 1]) {           /* одно сравнение */
            hold = a[i];                  /* одна перестановка */
            a[i] = a[i + 1];
            a[i + 1] = hold;
         }

   printf(" \nData items in ascending order\n");

   for (i = 0; i <= SIZE - 1; i++)
      printf("%4d" a[i]);

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

```
Data items in original order
   2   6   4   8  10  12  89  68  45  37
Data items in ascending order
   2   4   6   8  10  12  37  45  68  89
```

Рис. 6.15. Сортировка массива пузырьковым методом

Сначала программа сравнивает **a[0]** и **a[1]**, потом **a[1]** и **a[2]**, потом **a[2]** и **a[3]** и так далее, пока сравнение **a[8]** и **a[9]** не завершит проход. Обратите внимание, что хотя существует 10 элементов, выполняется только девять сравнений. Из-за способа проведения последовательных сравнений большое значение за один проход может переместиться в массиве на много позиций вниз, а небольшое значение может переместиться только на одну позицию вверх. За

первый проход самое большое значение гарантированно опускается в нижний элемент массива **a[9]**. За второй проход второе по величине значение гарантированно опускается в **a[8]**. За девятый проход девятое по величине значение оказывается в **a[1]**. В результате этого самое маленькое значение остается в **a[0]**, так что для сортировки массива необходимо только девять проходов, даже если он состоит из десяти элементов.

Сортировка выполняется посредством вложенного цикла **for**. В случае необходимости перестановки она выполняется путем трех присваиваний

```
hold = a[i];
a[i] = a[i + 1];
a[i + 1] = hold;
```

где в дополнительной переменной **hold** временно хранится одно из двух обмениваемых значений. Перестановка не может быть выполнена только двумя присваиваниями

```
a[i] = a[i + 1];
a[i + 1] = a[i];
```

Если, например, **a[i]** равно **7** и **a[i + 1]** равно **5**, то после первого присваивания оба значения будут равны **5** и значение **7** будет потеряно. Отсюда необходимость в дополнительной переменной **hold**.

Главным достоинством пузырьковой сортировки является то, что ее легко запрограммировать. Однако пузырьковая сортировка выполняется медленно. Это становится очевидным при сортировке больших массивов. В упражнениях мы разработаем более совершенные версии пузырьковой сортировки. В настоящее время уже разработаны гораздо более эффективные методы сортировки, чем пузырьковая. Позже мы исследуем некоторые из них. В углубленных курсах обучения программированию методы сортировки и поиска рассматриваются более полно.

6.7. Пример: вычисление среднего значения, медианы и наиболее вероятного значения с использованием массивов

Теперь рассмотрим более значительный пример. Компьютеры часто применяются для сбора и анализа результатов различных исследований и опросов общественного мнения. Программа на рис. 6.16 использует массив **response**, инициализированный 99-ю (см. символическую константу **SIZE**) ответами опроса. Каждый из ответов представлен числом от 1 до 9. Программа вычисляет среднее, медиану и наиболее вероятное из 99 значений.

Средним значением называется среднеарифметическое этих 99 значений. Функция **mean** вычисляет среднее значение путем суммирования 99 элементов и деления результата на 99.

Медианой называется «серединное значение». Функция **median** определяет медиану путем вызова функции **bubbleSort** для сортировки массива ответов в порядке возрастания и выбора среднего элемента **answer[SIZE / 2]** отсортированного массива. Имейте в виду, что в случае четного числа элементов медиана должна вычисляться как среднее значение двух «серединных» элементов. Функция **median** в настоящее время не обеспечивает такой возможности. Для вывода массива **response** вызывается функция **printArray**.

```
/* Эта программа производит анализ данных опроса.
   Она вычисляет среднее значение, медиану и наиболее вероятное
   значение */
#include <stdio.h>
#define SIZE 99

void mean(int []);
void median(int []);
void mode(int [], int []);
void bubbleSort(int []);
void printArray(int []);

main()
{
   int frequency[10] = {0},
       response[SIZE] = {6, 7, 8, 9, 8, 7, 8, 9, 8, 9,
                         7, 8, 9, 5, 9, 8, 7, 8, 7, 8,
                         6, 7, 8, 9, 3, 9, 8, 7, 8, 7,
                         7, 8, 9, 8, 9, 8, 9, 7, 8, 9,
                         6, 7, 8, 7, 8, 7, 9, 8, 9, 2,
                         7, 8, 9, 8, 9, 8, 9, 7, 5, 3,
                         5, 6, 7, 2, 5, 3, 9, 4, 6, 4,
                         7, 8, 9, 6, 8, 7, 8, 9, 7, 8,
                         7, 4, 4, 2, 5, 3, 8, 7, 5, 6,
                         4, 5, 6, 1, 6, 5, 7, 8, 7};
   mean(response);
   median(response);
   mode(frequency,response);
   return 0;
}

void mean(int answer[])
{
   int j, total = 0 ;

   printf("%s\n%s\n%s\n", "********", "  Mean", "********");

for (j=0; j<=SIZE-1; j++)
      total += answer[j];

   printf("The mean is the average value of the data\n"
          "items. The mean is equal to the total of\n"
          "all the data items devided by the number \n"
          "of data items (%d). The mean value for \n"
          "this run is : %d / %d = %.4f\n\n",
          SIZE, total, SIZE, (float) total / SIZE);
}

void median(int answer[])
{
   printf("\n%s\n%s\n%s\n%s",
          "********", "  Mean", "********",
          "The unsorted array of responses is");

   printArray(answer);
```

Рис. 6.16. Программа анализа данных опроса (часть 1 из 3)

```
   bubbleSort(answer);
   printf("\n\nThe sorted array is");
   printArray(answer);
   printf("\n\nThe median is element %d of \n"
          "the sorted %d element array.\n"
          "For this run the median is %d\n\n",
          SIZE / 2, SIZE, answer[SIZE / 2]);
}

void mode(int freq[], int answer[])
{
   int rating, j, h, largest=0, modeValue=0;

   printf("\n%s\n%s\n%s\n",
          "*********", "  Mode", "*********");

   for (rating=1; rating <= 9; rating++)
      freq[rating] = 0;

   for (j=0; j <= SIZE-1; j++)
      ++freq[answer[j]];

   printf("%s%11s%19s\n\n%54s\n%54s\n\n",
          "Response","Frequency","Histogram",
          "1   1   2   2", "5   0   5   0   5");

   for (rating=1; rating <= 9; rating++) {
      printf("%8d%11d      ", rating, freq[rating]);

      if (freq(rating) > largest) {
         largest = freq[rating];
         modeValue = rating;
      }

      for (h =1; h <= freq[rating]; h++)
         printf("*");

      printf("\n");
   }

   printf("The mode is the most frequent value.\n"
          "For this run the mode is %d which occured"
          " %d times.\n", modeValue, largest);
}

void bubbleSort(int a[])
{
   int pass, j, hold;

   for (pass=1; pass <= SIZE-1; pass++)

      for (j=0; j <= SIZE-2; j++)

         if (a[j] > a[j+1]) {
```

Рис. 6.16. Программа анализа данных опроса (часть 2 из 3)

```
            hold = a[j];
            a[j] = a[j+1];
            a[j+1] = hold;
         }
}
void printArray(int a[])
{
   int j;

   for (j=0; j <= SIZE-1; j++) {
      if (j % 20 == 0)
         printf("\n");

      printf("%2d", a[j]);
   }
}
```

Рис. 6.16. *Программа анализа данных опроса (часть 3 из 3)*

Наиболее вероятным значением называется значение, которое наиболее часто встречается среди 99 ответов. Функция **mode** определяет наиболее вероятное значение путем подсчета числа ответов каждого типа и выбора из них значения с наибольшей суммой. Эта версия функции **mode** не обрабатывает случай нескольких равных сумм (см. упражнение 6.14). Функция строит также гистограмму для помощи в определении наиболее вероятного значения графически. На рис. 6.17 содержится пробный прогон этой программы. Этот пример включает в себя большинство стандартных методик, обычно требующихся в задачах, связанных с массивами, в том числе передачу массива в функцию.

```
********
  Mean
********
The mean is the average value of the data
items. The Mean is equal to the total of
all the data items divided byy the number
of data items (99). The mean value
for this run is: 681 / 99 = 6.8788

********
 Median
********
The unsorted array of responses is
 6 7 8 9 8 7 8 9 8 9 7 8 9 5 9 8 7 8 7 8
 6 7 8 9 3 9 8 7 8 7 7 8 9 8 9 8 9 7 8 9
 6 7 8 7 8 7 9 8 9 2 7 8 9 8 9 8 9 7 5 3
 5 6 7 2 5 3 9 4 6 4 7 8 9 6 8 7 8 9 7 8
 7 4 4 2 5 3 8 7 5 6 4 5 6 1 6 5 7 8 7

The sorted array is
 1 2 2 2 3 3 3 3 4 4 4 4 4 5 5 5 5 5 5 5
 5 6 6 6 6 6 6 6 6 6 7 7 7 7 7 7 7 7 7 7
 7 7 7 7 7 7 7 7 7 7 7 7 7 8 8 8 8 8 8 8
 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8
 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9
```

Рис. 6.17. Пример выполнения программы анализа данных опроса (часть 1 из 2)

```
The median is element 49 of
the sorted 99 element array.
For this run the median is 7

********
  Mode
********
Response  Frequency           Histogram
                                      1    1    2    2
                                 5    0    5    0    5
       1          1           *
       2          3           ***
       3          4           ****
       4          5           *****
       5          8           ********
       6          9           *********
       7         23           ***********************
       8         27           ***************************
       9         19           *******************
The mode is the most frequent value.
For this run the mode is 8 which occurred 27 times.
```

Рис. 6.17. Пример выполнения программы анализа данных опроса (часть 2 из 2)

6.8. Поиск в массивах

Программисту часто приходится работать с большими объемами данных, хранимых в массивах. Может оказаться необходимым определить, содержит ли массив элемент, соответствующий некоторому *ключевому значению*. Процесс нахождения определенного элемента массива называется *поиском*. В этом разделе мы обсудим два метода поиска — простой метод *линейного поиска* (или *последовательного перебора*) и более эффективный метод двоичного поиска. В упражнениях 6.34 и 6.35 в конце этой главы вам будет предложено реализовать версии линейного и двоичного поиска с помощью рекурсии.

При линейном поиске (рис. 6.18) происходит сравнение каждого элемента массива с ключом поиска. Поскольку массив не упорядочен особым образом, вероятность нахождения требуемого значения в первом и в последнем элементах массива одинакова. Таким образом, в среднем программа должна будет сравнить ключ поиска с половиной элементов массива.

Метод линейного поиска хорошо работает для небольших или несортированных массивов. Однако для больших массивов он неэффективен. В случае сортированного массива может быть использован высокоскоростной метод двоичного поиска.

При двоичном алгоритме поиска после каждого сравнения исключается половина элементов массива, в котором проводится поиск. Алгоритм находит средний элемент массива и сравнивает его с ключом поиска. Если они равны, ключ поиска считается найденным и возвращается индекс этого элемента. Если они не равны, задача упрощается до поиска в одной половине массива.

Если ключ поиска меньше среднего элемента массива, поиск производится в первой половине массива, в противном случае поиск производится во второй половине. Если ключ поиска в указанном подмассиве (части первоначаль-

ного массива) не найден, алгоритм повторяется для четверти массива. Поиск продолжается до тех пор, пока ключ не окажется равен среднему элементу подмассива или пока подмассив не будет состоять из одного элемента, не равного ключу (это означает, что ключ поиска не найден).

```
/* Последовательный перебор массива */
#include <stdio.h>
#define SIZE 100

int linearSearch(int [], int, int);

main()
{
   int a[SIZE], x, searchKey, element;

   for (x = 0; x <= SIZE - 1; x++) /* порождает какие-то данные */
      a[x] = 2 * x;

   printf("Enter integer search key:\n");
   scanf("%d", &searchKey);
   element = int linearSearch(a, searchKey, SIZE);

   if (element != -1)
      printf("Found value in element %d\n", element);
   else
      printf("Value not found\n");

   return 0;
}

int linearSearch(int array[], int key, int size)
{
   int n;

   for (n = 0; n <= size - 1; n++)
      if (array[n] == key)
         return n;

   return -1
}
```

```
Enter integer search key:
36
Found value in element 18

Enter integer search key:
37
Value not found
```

Рис. 6.18. Последовательный перебор массива

В наихудшем случае двоичный поиск в массиве из 1024 элементов потребует только 10 сравнений. Последовательное деление 1024 на 2 дает значения 512, 256, 128, 64, 32, 16, 8, 4, 2 и 1. Число 1024 (2^{10}) до получения значения, равного 1, делится на 2 только десять раз. Деление на 2 соответствует одному сравнению в алгоритме двоичного поиска. Чтобы найти ключ в массиве из 1048576 (2^{20}) элементов, потребуется максимум 20 сравнений. В массиве из

миллиарда элементов потребуется максимум 30 сравнений. Это является огромным увеличением эффективности в сравнении с последовательным перебором, который требует в среднем сравнения ключа поиска с половиной элементов массива. Для массива из миллиарда элементов разница между 500 миллионами сравнений (в среднем) и максимум 30 сравнениями очевидна! *Максимальное число сравнений для любого массива можно определить* путем нахождения первой степени 2, превосходящей число элементов в массиве.

На рис. 6.19 представлена итеративная версия функции **binarySearch**. Функция принимает четыре параметра — целочисленный массив **b**, целочисленную переменную **searchKey**, индекс массива **low** и индекс массива **high**. Если ключ поиска не соответствует среднему элементу подмассива, один из индексов **low** или **high** изменяется таким образом, чтобы поиск можно было проводить в меньшем подмассиве. Если ключ меньше среднего элемента, индексу **high** присваивается значение **middle - 1** и поиск продолжается для элементов от **low** до **middle - 1**. Если ключ больше среднего элемента, индексу **low** присваивается значение **middle + 1** и поиск продолжается для элементов от **middle + 1** до **high**. В программе используется массив из 15 элементов. Первая степень 2, превосходящая число элементов в этом массиве, равна 16 (2^4), поэтому для нахождения ключа потребуется максимум 4 сравнения. В программе используются функции **printHeader** для вывода индексов массива и **printRow** для вывода каждого подмассива, возникающего в процессе двоичного поиска. Средний элемент в каждом подмассиве отмечен звездочкой (*), указывающей на элемент, с которым сравнивается ключ поиска.

```
/* Двоичный поиск в массиве */

#include <stdio.h>
#define SIZE 15

int binarySearch(int [], int, int, int);
void printHeader(void);
void printRow(int [], int, int, int);

main()
{
   int a[SIZE], i, key, result;

   for (i = 0; i <= SIZE - 1; i++)
      a[i] = 2 * i;

   printf("Enter a number between 0 and 28: ");
   scanf("%d", &key);

   printHeader();
   result = binarySearch(a, key, 0, SIZE - 1);

   if (result != -1)
      printf("\n%d found in array element %d\n", key, result);
   else
      printf("\n%d not found\n", key);

   return 0;
}
```

Рис. 6.19. Двоичный поиск в сортированном массиве (часть 1 из 3)

```
int binarySearch(int b[], int searchKey, int low, int high)
{
   int middle;

   while (low <= high) {
      middle = (low + high) / 2;

      printRow(b, low, middle, high);

      if (searchKey == b[middle])
         return middle;
      else if (searchKey < b[middle])
         high = middle - 1;
      else
         low = middle + 1;
   }

   return -1;      /* ключ поиска не найден */
}

/* Печатает заголовок для выводимых данных */
void printHeader(void)
{
   int i;

   printf("\nSubscripts:\n");

   for (i = 0; i <= SIZE - 1; i++)
      printf("%3d ", i);

   printf("\n");

   for (i = 1; i <= 4 * SIZE; i++)
      printf("-");

   printf("\n");
}

/* Выводит строку данных, показывающих часть массива,
   обрабатываемую в настоящее время. */
void printRow(int b[], int low, int mid, int high)
{
   int i;

   for (i = 0; i <= SIZE - 1; i++)
      if (i < low || i > high)
         printf("    ");
      else if (i == mid)
         printf("%3d*", b[i]);   /* отмечает среднее значение */
      else
         printf("%3d ", b[i]);

   printf("\n");
}
```

Рис. 6.19. Двоичный поиск в сортированном массиве (часть 2 из 3)

```
Enter a number between 0 and 28: 25
Subscripts:
  0   1   2   3   4   5   6   7   8   9  10  11  12  13  14
---------------------------------------------------------------
  0   2   4   6   8  10  12  14* 16  18  20  22  24  26  28
                                 16  18  20  22* 24  26  28
                                                 24  26* 28
                                                 24*

25 not found

Enter a number between 0 and 28: 8
Subscripts:
  0   1   2   3   4   5   6   7   8   9  10  11  12  13  14
---------------------------------------------------------------
  0   2   4   6   8  10  12  14* 16  18  20  22  24  26  28
  0   2   4   6*  8  10  12
                  8  10* 12
                  8*

8 found in array element 4

Enter a number between 0 and 28: 6
Subscripts:
  0   1   2   3   4   5   6   7   8   9  10  11  12  13  14
---------------------------------------------------------------
  0   2   4   6   8  10  12  14* 16  18  20  22  24  26  28
  0   2   4   6*  8  10  12

6 found in array element 3
```

Рис. 6.19. Двоичный поиск в сортированном массиве (часть 3 из 3)

6.9. Многомерные массивы

Массивы в C могут иметь несколько индексов. Многомерные массивы часто применяются для представления *таблиц*, состоящих из значений, упорядоченных по *строкам* и *столбцам*. Для идентификации конкретного элемента таблицы мы должны указать два индекса: первый, идентифицирующий (по соглашению) строку элемента, и второй, идентифицирующий столбец. Таблицы, или массивы, требующие двух индексов для идентификации данного элемента, называются *двумерными массивами*. Обратите внимание, что многомерные массивы могут иметь более двух индексов. Стандартом ANSI установлено, что система ANSI C должна поддерживать по крайней мере 12 индексов массива.

Рис. 6.20 иллюстрирует двумерный массив **a**. Массив содержит три строки и четыре столбца, поэтому говорят, что это массив 3 на 4. В общем случае, массив с m строками и n столбцами называется *массивом m на n*.

На рис. 6.20 каждый элемент массива **a** идентифицируется как **a[i][j]**; здесь **a** имя массива, **i** и **j** его индексы, которые однозначно определяют каждый элемент в массиве **a**. Обратите внимание, что у всех элементов первой строки массива первый индекс равен **0**; у всех элементов четвертого столбца массива второй индекс равен **3**.

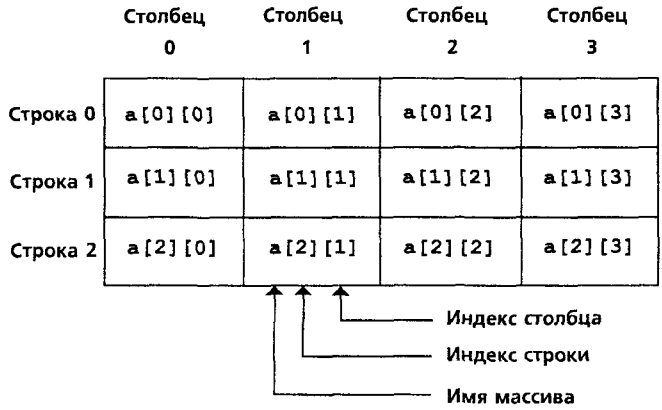


Рис. 6.20. Двумерный массив с тремя строками и четырьмя столбцами

Распространенная ошибка программирования 6.9

Неправильная ссылка на элемент двумерного массива **a[x][y]** в виде **a[x,y]**.

Многомерный массив, подобно одномерному, может быть инициализирован при его объявлении. Например, двумерный массив **b[2][2]** может быть объявлен и инициализирован посредством

```
int b[2][2] = {{1, 2}, {3, 4}};
```

Значения группируются в фигурных скобках по строкам. Таким образом, **1** и **2** инициализируют **b[0][0]** и **b[0][1]**, **3** и **4** инициализируют **b[1][0]** и **b[1][1]**. Если для данной строки недостаточно инициализирующих значений, оставшиеся ее элементы инициализируются нулями. Таким образом, объявление

```
int b[2][2] = {{1}, {3,4}};
```

инициализировало бы **b[0][0]** единицей, **b[0][1]** нулем, **b[1][0]** тройкой и **b[1][1]** четверкой.

Рис. 6.21 демонстрирует инициализацию двумерных массивов при их объявлении. В программе объявляются три массива с двумя строками и тремя столбцами (каждый состоит из шести элементов). Объявление массива **array1** предусматривает шесть инициализирующих значений в двух подсписках. Первый подсписок инициализирует первую строку массива значениями 1, 2 и 3; второй подсписок инициализирует вторую строку массива значениями 4, 5 и 6. Если из списка инициализирующих значений массива **array1** убрать фигурные скобки, в которые заключен каждый подсписок, компилятор автоматически проинициализирует элементы первой строки, а затем элементы второй строки. Объявление массива **array2** предусматривает пять инициализирующих значений. Инициализирующие значения присваиваются сначала первой строке, затем второй строке. Все элементы, для которых нет явного инициализирующего значения, автоматически инициализируются нулями, поэтому **array2[1][2]** инициализирован 0. Объявление массива **array3** предусматривает три инициализирующих значения в двух подсписках. Подсписок для первой строки явно инициализирует первые два элемента первой строки единицей и двойкой. Третий элемент автоматически инициализируется нулем. Под-

список для второй строки явно инициализирует первый элемент четверкой. Последние два элемента автоматически инициализируются нулями.

```
/* Инициализация многомерного массива */
#include <stdio.h>

void printArray(int [][3]);

main()
{
   int array1[2][3] = { {1, 2, 3}, {4, 5, 6} },
   array2[2][3] = { 1, 2, 3, 4, 5 },
   array3[2][3] = { {1, 2}, {4} };

   printf("Values in array1 by row are:\n");
   printArray(array1);

   printf("Values in array2 by row are:\n");
   printArray(array2);

   printf("Values in array3 by row are:\n");
   printArray(array3);

   return 0;
}

void printArray(int a[][3])
{
   int i,j;

   for (i = 0; i <= 1; i++) {

      for (j = 0; j <= 2; j++)
         printf("%d ", a[i][j]);

      printf("\n");
   }
}
```

```
Values in array1 by row are:
1 2 3
4 5 6
Values in array2 by row are:
1 2 3
4 5 0
Values in array3 by row are:
1 2 0
4 0 0
```

Рис. 6.21. Инициализация многомерных массивов

Для вывода элементов каждого массива в программе вызывается функция **printArray**. Обратите внимание, что в определении функции параметр-массив специфицирован как **int a[][3]**. Когда функция принимает в качестве параметра одномерный массив, в списке параметров функции скобки, относящиеся к массиву, остаются незаполненными. Первый индекс многомерного массива

также не требуется, однако все последующие индексы необходимы. Компилятор использует эти индексы для определения местонахождения в памяти элементов многомерных массивов. Все элементы массива хранятся в памяти последовательно, независимо от количества индексов. В двумерном массиве за первой строкой в памяти следует вторая строка.

Указание значений индексов при объявлении параметров функции дает возможность компилятору сообщить ей о способе отыскания элементов в массиве. В двумерных массивах каждая строка по существу является одномерным массивом. Для определения местонахождения элемента в конкретной строке компилятор должен знать, сколько в точности элементов находится в каждой строке, чтобы при обращении к этому массиву он мог пропустить соответствующее число блоков памяти. Таким образом, при обращении к элементу **a[1][2]** в нашем примере компилятор знает, что для того, чтобы добраться до второй строки (строка 1), ему нужно пропустить в памяти три элемента первой строки.

После этого компилятор обращается к третьему элементу этой строки (элемент 2).

Во многих манипуляциях с массивами обычно используется структура повторения **for**. Например, следующий оператор устанавливает все элементы в третьей строке массива **a**, изображенного на рис. 6.20, в нуль:

```
for (column = 0; column <= 3; column++)
   a[2][column] = 0;
```

Мы устанавливаем *третью* строку, следовательно, мы знаем, что первый индекс всегда равен **2** (**0** это первая строка, **1** — вторая строка). В цикле **for** изменяется только второй индекс (т.е. индекс столбца). Предыдущая структура **for** эквивалентна четырем операторам присваивания:

```
a[2][0] = 0;
a[2][1] = 0;
a[2][2] = 0;
a[2][3] = 0;
```

Следующая вложенная структура **for** подсчитывает сумму всех элементов массива **a**.

```
total = 0;

for (row = 0; row <= 2; row++)

   for (column = 0; column <= 3; column++)
      total += a[row][column];
```

В этой структуре **for** элементы массива суммируются построчно. Выполнение внешней структуры **for** начинается с установки **row** (т.е. индекса строки) в **0**, так что при помощи внутренней структуры **for** могут быть просуммированы элементы первой строки. Во внешней структуре **for** переменная **row** увеличивается до **1**, так что могут быть просуммированы элементы второй строки. Затем во внешней структуре **for** переменная **row** увеличивается до **2**, и суммируются элементы третьей строки.

В программе на рис. 6.22 с использованием структур **for** над массивом **studentGrades** размером 3 на 4 выполняются некоторые другие манипуляции, обычные для массивов. Каждая строка массива представляет студента, а каждый столбец представляет оценку за один из четырех экзаменов, которые студенты сдавали в семестре. Манипуляции с массивами выполняются посредст-

вом четырех функций. Функция **minimum** определяет самую низкую оценку среди всех студентов за семестр. Функция **maximum** определяет самую высокую. Функция **average** определяет для данного студента его среднюю оценку за семестр. Функция **printArray** выводит двумерный массив в наглядной табличной форме.

```
/* Пример двумерного массива */
#include <stdio.h>
#define STUDENTS 3
#define EXAMS 4

int minimum(int [][EXAMS], int, int);
int maximum(int [][EXAMS], int, int);
float average(int [], int);
void printArray(int [][EXAMS], int, int);

main()
{
   int student,
   studentGrades[STUDENTS][EXAMS] = {{77, 68, 86, 73},
                                     {96, 87, 89, 78},
                                     {70, 90, 86, 81}};

   printf("The array is:\n");
   printArray(studentGrades, STUDENTS, EXAMS);
   printf("\n\nLowest grade: %d\nHighest grade: %d\n",
   minimum(studentGrades, STUDENTS, EXAMS),
   maximum(studentGrades, STUDENTS, EXAMS));

   for (student = 0; student <= STUDENTS - 1; student++)
      printf("The average grade for student %d is %.2f\n",
             student, average(studentGrades[student], EXAMS));

   return 0;
}

/* Находит минимальную оценку */
int minimum(int grades[][EXAMS], int pupils, int tests)
{
   int i, j, lowGrade = 100;

   for (i = 0; i <= pupils - 1; i++)
      for (j = 0; j <= tests - 1; j++)
         if (grades[i][j] < lowGrade)
            lowGrade = grades[i][j];

   return lowGrade;
}

/* Находит максимальную оценку */
int maximum(int grades[][EXAMS], int pupils, int tests)
{
   int i, j, highGrade = 0;
```

Рис. 6.22. Пример использования двумерных массивов (часть 1 из 2)

```
   for (i = 0; i <= pupils - 1; i++)
      for (j = 0; j <= tests - 1; j++)
         if (grades[i][j] < highGrade)
            highGrade = grades[i][j];

   return highGrade;
}

/* Определяет среднюю оценку для данного студента */
float average(int setOfGrades[], int tests)
{
   int i, total = 0;

   for (i = 0; i <= tests - 1; i++)
      total += setOfGrades[i];

   return (float) total / tests;
}

/* Выводит массив */
void printArray(int grades[][EXAMS], int pupils, int tests)
{
   int i, j;

   printf("              [0]  [1]  [2]  [3]");

   for (i = 0; i <= pupils - 1; i++) {
      printf("\nstudentGrades[%d] ", i);

      for (j = 0; j <= tests - 1; j++)
         printf("%-5d", grades[i][j]);
   }
}
```

```
The array is:
                [0]   [1]   [2]   [3]
studentGrades[0] 77    68    86    73
studentGrades[1] 96    87    89    78
studentGrades[2] 70    90    86    81

Lowest grade: 68
Highest grade: 96
The average grade for student 0 is 76.00
The average grade for student 0 is 87.50
The average grade for student 0 is 81.75
```

Рис. 6.22. *Пример использования двумерных массивов (часть 2 из 2)*

Каждая из функций **minimum**, **maximum** и **printArray** принимает три параметра — массив **studentGrades** (называемый в каждой из функций **grades**), количество студентов (строк массива) и количество экзаменов (столбцов массива). В каждой из функций по массиву **grades** выполняется цикл с вложенными структурами **for**. Следующая вложенная структура является фрагментом определения функции **minimum**:

```
for (i = 0; i <= pupils - 1; i++)
   for (j = 0; j <= tests - 1; j++)
      if (grades[i][j] < lowGrade)
         lowGrade = grades[i][j];
```

Выполнение внешней структуры **for** начинается с установки переменной **i** (т.е. индекса строки) в **0**, так что в теле внутренней структуры **for** элементы первой строки могут сравниваться с переменной **lowGrade**. Во внутренней структуре происходит выполнение цикла для четырех оценок данной строки и сравнение каждой оценки с **lowGrade**. Если оценка меньше, чем **lowGrade**, переменной **lowGrade** присваивается значение этой оценки. Затем во внешней структуре индекс строки увеличивается до **1**. С переменной **lowGrade** сравниваются элементы второй строки. Затем индекс строки увеличивается до **2**, и с переменной **lowGrade** сравниваются элементы третьей строки. Когда выполнение всей структуры заканчивается, в **lowGrade** содержится наименьшая оценка в двумерном массиве. Функция **maximum** работает аналогично функции **minimum**.

Функция **average** принимает два аргумента — одномерный массив оценок данного студента с именем **setOfGrades** и число элементов (экзаменов), содержащихся в этом массиве. При вызове функции **average** первым передаваемым параметром является **studentGrades[student]**. Это приводит к тому, что в функцию передается адрес одной строки двумерного массива. Параметр **studentGrades[1]** представляет собой начальный адрес второй строки массива. Не забывайте, что двумерный массив по существу является массивом одномерных массивов и что имя одномерного массива является адресом этого массива в памяти. Функция вычисляет сумму элементов массива, делит общий итог на число экзаменов и возвращает результат в виде числа с плавающей точкой.

Резюме

- В языке С списки значений хранятся в массивах. Массив является группой логически связанных ячеек памяти. Их связывает то, что все они имеют одно и то же имя и один и тот же тип. Для обращения к конкретной ячейке, или элементу массива, мы указываем имя массива и индекс.
- Индекс может быть целым числом или целочисленным выражением. Если в качестве индекса используется выражение, то для определения конкретного элемента массива происходит оценка этого выражения.
- Важно обратить внимание на различие между ссылкой на седьмой элемент массива и на элемент массива номер семь. Для седьмого элемента индекс равен **6**, в то время как элемент массива номер семь имеет индекс **7** (фактически это восьмой элемент массива). Это является источником ошибок преждевременного выхода из цикла.
- Массивы занимают место в памяти. Чтобы зарезервировать 100 элементов для целочисленного массива **b** и 27 элементов для целочисленного массива **x**, программист пишет

  ```
  int b[100], x[27];
  ```

- Массив типа **char** может использоваться для хранения символьных строк.

- Элементы массива могут быть инициализированы тремя способами: посредством объявления, присваивания и ввода.
- Если инициализирующих значений в объявлении меньше, чем элементов в массиве, С автоматически инициализирует оставшиеся элементы нулями.
- С допускает ссылки на элементы за пределами массива.
- Для инициализации символьного массива может быть использован строковый литерал.
- *Все строки в языке С заканчиваются нулевым символом.* Представлением нулевого символа в виде символьной константы является **'\0'**.
- Символьные массивы могут быть инициализированы символьными константами в списке инициализирующих значений.
- К отдельным символам строки, хранимой в массиве, можно обращаться непосредственно, пользуясь для этого индексной нотацией.
- Строка может быть непосредственно введена в символьный массив с клавиатуры при помощи функции **scanf** и спецификации преобразования **%s**.
- Символьный массив, представляющий строку, может быть выведен с помощью функции **printf** и спецификации преобразования **%s**.
- Применяйте при локальном объявлении массива ключевое слово **static**, с тем чтобы массив не создавался при каждом вызове функции и не разрушался при каждом выходе из нее.
- Массивы, объявленные с ключевым словом **static**, автоматически инициализируются один раз во время компиляции. Если программист не инициализирует статический массив явно, он инициализируется компилятором нулями.
- Для передачи массива в функцию передается его имя. Для передачи в функцию одного элемента массива просто передайте имя массива с последующим индексом данного элемента (в квадратных скобках).
- В языке С передача массива в функцию происходит путем имитации передачи параметра по ссылке — вызываемые функции могут изменять значения элементов в подлинных массивах вызывающих программ. Имя массива фактически является адресом первого элемента массива! Поскольку происходит передача начального адреса массива, вызываемая функция точно знает, где хранится массив.
- Для того, чтобы функция при обращении к ней могла принять массив, в списке параметров этой функции должно быть указано, что ожидается передача массива. Размер массива в скобках при этом не требуется.
- Спецификация преобразования **%p** обычно служит для вывода адресов в виде шестнадцатеричных чисел.
- Для предотвращения изменения функцией значений элементов массива в языке С предусмотрен специальный модификатор типа **const**. Когда параметру-массиву предшествует **const**, элементы массива становятся константами в теле функции и любая ее попытка изменить элемент массива приводит к ошибке времени компиляции.

- Массив может быть сортирован с применением метода пузырьковой сортировки. Делается несколько проходов по массиву. При каждом проходе сравниваются последовательные пары элементов. Если элементы расположены по порядку (или если их значения совпадают), ничего не меняется. Если элементы пары расположены не по порядку, их значения меняются местами. Для небольших массивов пузырьковая сортировка приемлема, но для больших она является неэффективной в сравнении с другими, более сложными алгоритмами сортировки.
- При линейном поиске (последовательном переборе) происходит сравнение каждого элемента массива с ключевым значением. Поскольку массив не упорядочен особым образом, вероятность нахождения требуемого значения в первом и в последнем элементах массива одинакова. Таким образом, в среднем программа должна будет сравнить ключ поиска с половиной элементов массива. Метод последовательного перебора хорошо работает для небольших или несортированных массивов.
- При двоичном алгоритме поиска после каждого сравнения исключается половина элементов массива, в котором проводится поиск. Алгоритм находит средний элемент массива и сравнивает его с ключом поиска. Если они равны, ключ поиска считается найденным и возвращается индекс этого элемента в массиве. Если они не равны, задача упрощается до поиска в одной половине массива.
- В наихудшем случае двоичный поиск в массиве из 1024 элементов потребует только 10 сравнений. Чтобы найти ключевое значение в массиве из 1048576 (2^{20}) элементов, потребуется максимум 20 сравнений. В массиве из миллиарда элементов потребуется максимум 30 сравнений.
- Массивы могут использоваться для представления таблиц, состоящих из значений, упорядоченных по строкам и столбцам. Для указания конкретного элемента таблицы определяются два индекса: первый, идентифицирующий (по соглашению) строку, в которой содержится элемент, и второй, идентифицирующий столбец. Таблицы, или массивы, требующие для идентификации элемента двух индексов, называются двумерными массивами.
- Стандартом установлено, что система ANSI C должна поддерживать по крайней мере 12 индексов массива.
- Многомерный массив может быть инициализирован при его объявлении списком инициализирующих значений.
- Когда функция принимает в качестве параметра одномерный массив, в списке параметров функции скобки, относящиеся к массиву, остаются незаполненными.
- Первый индекс многомерного массива также не требуется, однако все последующие индексы необходимы. Компилятор использует эти индексы для определения местонахождения в памяти элементов многомерных массивов.
- Для передачи одной строки двумерного массива в функцию, принимающую одномерный массив, просто передайте имя массива с последующим первым индексом.

Терминология

a[i]
a[i][j]
анализ данных опроса
временная область для обмена значений
выражение в качестве индекса
выход за пределы массива
гистограмма
двойная точность
двумерный массив
директива препроцессора **#define**
заменяющий текст
значение элемента
имя массива
индекс
индекс столбца
индекс строки
инициализация массива
квадратные скобки
ключ поиска
линейный поиск
массив
массив n на m
масштабируемость
медиана
многомерный массив
наиболее вероятное значение
номер позиции
нулевой элемент
нуль-символ '\0'
общая сумма элементов массива
объявление массива
одномерный массив
ошибка смещения индекса
передача массивов в функции
передача по ссылке
поиск в массиве
проверка на выход за пределы массива
проход сортировки
пузырьковая сортировка
символическая константа
скаляр
скалярная величина
сортировка
сортировка погружением
сортировка элементов массива
спецификатор преобразования %**p**
список инициализации массива
среднее
столбцовая диаграмма
строка
таблица значений
табличная форма
трехмерный массив
элемент массива

Распространенные ошибки программирования

6.1. Важно обратить внимание на различие между «седьмым элементом массива» и «элементом массива номер семь». Поскольку индексы массива начинаются с 0, «седьмой элемент массива» имеет индекс 6, в то время как «элемент массива номер семь» имеет индекс 7 и фактически является восьмым элементом массива. Это источник так называемых ошибок смещения индекса.

6.2. Отсутствие инициализации массива, элементы которого должны быть инициализированы.

6.3. Указание в списке инициализации большего числа значений, чем количество элементов в массиве, является синтаксической ошибкой.

6.4. Завершение директив препроцессора **#define** или **#include** точкой с запятой. Не забывайте, что директивы препроцессора не являются операторами языка C.

6.5. Присвоение значения символической константе в выполняемом операторе является синтаксической ошибкой. Символическая константа не является переменной. Компилятор не резервирует для нее места в памяти, как в случае переменных, содержащих значения во время выполнения программы.

6.6. Ссылка на элемент за пределами массива.

6.7. Передача функции **scanf** символьного массива, недостаточно большого для хранения вводимой с клавиатуры строки, может привести к потере данных в программе и другим ошибкам времени выполнения.

6.8. Предположение, что элементы локального массива, объявленного как **static**, инициализируются нулями всякий раз при вызове функции, где этот массив объявлен.

6.9. Неправильная ссылка на элемент двумерного массива **a[x][y]** в виде **a[x,y]**.

Хороший стиль программирования

6.1. Используйте для имен символических констант только символы верхнего регистра. Это выделяет эти константы в тексте программы и напоминает программисту, что символические константы не являются переменными.

6.2. Стремитесь к ясности программы. Иногда имеет смысл отказаться от наиболее эффективного использования памяти или процессорного времени в пользу написания более понятного кода.

6.3. При выполнении цикла над элементами массива его индекс никогда не должен быть меньше 0 и должен быть строго меньше общего количества элементов, т.е. не больше, чем (размер массива - 1). Убедитесь, что условие завершения цикла не допускает обращения к элементам вне этого диапазона.

6.4. Указывайте в структуре **for** наибольшее значение индекса массива, способствуя тем самым устранению ошибок преждевременного выхода из цикла.

6.5. Программы должны проверять правильность всех входных значений, чтобы не допустить воздействия ошибочной информации на вычисления программы.

6.6. Некоторые программисты включают имена переменных в прототипы функций для того, чтобы сделать программы более понятными. Компилятор игнорирует эти имена.

Советы по повышению эффективности

6.1. Иногда соображения эффективности намного перевешивают соображения понятности.

6.2. Последствия (обычно серьезные) ссылок на элементы за пределами массива являются системно-зависимыми.

6.3. В функциях, содержащих локальные массивы и часто входящих и выходящих из области действия, делайте массивы статическими, чтобы они не создавались заново при каждом вызове функции.

6.4. Передача массивов посредством имитации передачи по ссылке имеет смысл из соображениям эффективности. Если бы массивы передавались по значению, передавалась бы копия каждого элемента. Для больших часто передаваемых массивов это было бы расточительно

по времени и приводило бы к расходу значительного объема памяти для хранения их копий.

6.5. Часто наиболее простые алгоритмы работают плохо. Их достоинством является простота написания, тестирования и отладки. Однако для достижения максимальной эффективности часто требуются более сложные алгоритмы.

Общие методические замечания

6.1. Определение размера всех массивов с помощью символических констант повышает общность программ (делает их масштабируемыми).

6.2. Возможна передача массива по значению с помощью простого приема, который описывается в главе 10.

6.3. К параметру-массиву в определении функции может применяться модификатор **const**, чтобы не допустить изменения в теле функции исходного массива. Это является еще одним примером принципа наименьших привилегий. Функциям не следует давать возможности изменять массивы, если только это не является совершенно необходимым.

Упражнения для самоконтроля

6.1. Заполните пропуски в каждом из следующих утверждений:

a) Списки и таблицы значений хранятся в _____.

b) Элементы массива связаны в том отношении, что они имеют одни и те же _____ и _____.

c) Число, используемое для обращения к конкретному элементу массива, называется _____.

d) Для объявления размера массива должна использоваться _____, поскольку в этом случае программа становится более общей.

e) Процесс размещения элементов массива в определенном порядке называется _____ массива.

f) Процесс определения, содержит ли массив некоторое ключевое значение, называется _____ в массиве.

g) Массив, который имеет два индекса, называется _____ массивом.

6.2. Установите, являются ли следующие высказывания верными или невнрными. Если высказывание неверно, объясните почему.

a) В массиве может храниться много различных типов значений.

b) Индексом массива может быть число типа **float**.

c) Если число инициализирующих значений в списке инициализации меньше числа элементов массива, С автоматически инициализирует оставшиеся элементы последним значением в списке инициализации.

d) Если список инициализации содержит больше инициализирующих значений, чем имеется элементов в массиве, это является ошибкой.

e) Отдельный элемент массива, который передается в функцию и который изменяется в вызываемой функции, в вызывающей функции будет содержать измененное значение.

6.3. Выполните следующие действия для массива с именем **fractions**.

a) Определите символическую константу **SIZE**, которая будет замещена на текст 10.

b) Объявите массив из **SIZE** элементов типа **float** и инициализируйте элементы нулями.

c) Назовите четвертый элемент от начала массива.

d) Обратитесь к элементу массива номер 4.

e) Присвойте значение **1.667** элементу массива номер девять.

f) Присвойте значение **3.333** седьмому элементу массива.

g) Выведите элементы массива с номерами 6 и 9 с точностью до двух знаков справа от десятичной точки и покажите выводимые значения, которые фактически отображаются на экране.

h) Выведите все элементы массива, используя структуру повторения **for**. Сделайте при этом предположение, что в качестве управляющей переменной цикла была определена целая переменная **x**. Покажите выводимые значения.

6.4. Выполните следующие действия для массива с именем **table**.

a) Объявите целочисленный массив с 3 строками и 3 столбцами. Предположите, что была определена символическая константа **SIZE**, равная 3.

b) Сколько элементов содержится в массиве?

c) Используйте структуру повторения **for** для инициализации каждого элемента массива суммой его индексов. Предположите, что в качестве управляющих переменных объявлены целые переменные **x** и **y**.

d) Выведите значения каждого элемента массива **table**. Предположите, что массив был инициализирован с помощью объявления

```
int table[SIZE][SIZE] = {{1, 8}, {2, 4, 6}, {5}};
```

и целые переменные **x** и **y** объявлены в качестве управляющих переменных. Покажите выводимые значения.

6.5. Найдите ошибку в каждом из следующих фрагментов программы и исправьте ее.

a)
```
#define SIZE 100;
```

b)
```
SIZE = 10;
```

c) Предположим, что `int b[10] = {0}, i;`
```
for (i = 0; i <= 10; I++)
   b[i] = 1;
```

d)
```
#indlude <stdio.h>;
```

e) Предположим, что `int a[2][2] = {{1, 2}, {3, 4}};`
```
a[1,1] = 5;
```

Ответы на упражнения для самоконтроля

6.1. a) массивах. b) имя, тип. c) индексом. d) символическая константа. e) сортировкой. f) поиском. g) двумерным.

6.2. a) Неверно. В массиве могут храниться только значения одного и того же типа.

b) Неверно. Индексом массива должно быть целое число или целочисленное выражение.

c) Неверно. С автоматически инициализирует оставшиеся элементы нулями.

d) Верно.

e) Неверно. Отдельные элементы массива передаются по значению. Если же в функцию передается весь массив, то все изменения будут отражены в оригинале.

6.3. a)
```
#define SIZE 10
```

b)
```
float fractions [SIZE] = {0};
```

c)
```
fractions [3]
```

d)
```
fractions [4]
```

e)
```
fractions [9] = 1.667;
```

f)
```
fractions [6] = 3.333;
```

g)
```
printf("%.2f %.2f\n", fractions[6], fractions[9]);
```

Вывод: 3.33 1.67.

h)
```
for (x = 0; x <= SIZE - 1; x++)
     printf ( "fractions[%d] = %f\n", x, fractions[x]);
```

Вывод:
```
fractions[0] = 0.000000
fractions[1] = 0.000000
fractions[2] = 0.000000
fractions[3] = 0.000000
fractions[4] = 0.000000
fractions[5] = 0.000000
fractions[6] = 3.333000
fractions[7] = 0.000000
fractions[8] = 0.000000
fractions[9] = 1.667000
```

6.4. a)
```
int table[SIZE][SIZE];
```

b) Девять элементов.

c)
```
for (x = 0; x <= SIZE - 1; x++)
     for (y = 0; y <= SIZE - 1; y++)
        table[x][y] = x + y;
```

d)
```
for (x = 0; x <= SIZE - 1; x++)
     for (y = 0; y <= SIZE - 1; y++)
        printf("table[%d][%d] = %d\n", x, y, table[x]ly]);
```

Вывод:
```
table[0][0] = 1
table[0][1] = 8
```

```
table[0][2] = 0
table[1][0] = 2
table[1][1] = 4
table[1][2] = 6
table[2][0] = 5
table[2][1] = 0
table[2][2] = 0
```

6.5. a) Ошибка: точка с запятой в конце директивы препроцессора **#define**.

Исправление: уберите точку с запятой.

b) Ошибка: присвоение значения символической константе с использованием операции присваивания.

Исправление: присвойте значение символической константе в директиве препроцессора **#define** без использования операции присваивания, как в **#define SIZE 10**.

c) Ошибка: ссылка на элемент массива за его пределами (**b[10]**).

Исправление: измените верхнее значение управляющей переменной на **9**.

d) Ошибка: точка с запятой в конце директивы препроцессора **#include**.

Исправление: уберите точку с запятой.

e) Ошибка: некорректное использование индексов массива.

Исправление: замените оператор на **a[1][1] = 5;**

Упражнения

6.6. Заполните пробелы в каждом из следующих предложений:

a) В языке C списки значений хранятся в _____.

b) Элементы массива связаны в том отношении, что они _____.

c) При ссылке на элемент массива номер его позиции, содержащийся в скобках, называется _____.

d) Именами пяти элементов массива **p** являются _____, _____, _____, _____ и _____.

e) Содержимое конкретного элемента массива называется _____ этого элемента.

f) Присвоение имени массиву, задание его типа и определения числа его элементов называется _____ массива.

g) Процесс размещения элементов массива в возрастающем или убывающем порядке называется _____.

h) В двумерном массиве первый индекс идентифицирует (по соглашению) _____ элемента, а второй индекс — его _____.

i) Массив m на n содержит _____ строк, _____ столбцов и _____ элементов.

j) Именем элемента массива **d** в строке 3 и столбце 5 является _____.

6.7. Установите, какие из следующих высказываний являются верными, а какие неверными; для неверных высказываний объясните, почему.

a) Для обращения к данному участку памяти, или элементу, внутри массива мы указываем имя массива и значение конкретного элемента.

b) При объявлении массива для него резервируется память.

c) Чтобы указать, что для целочисленного массива **p** должно быть зарезервировано 100 участков памяти, программист пишет объявление

```
p[100];
```

d) Программа на C, инициализирующая элементы 15-элементного массива нулями, должна содержать один оператор **for**.

e) Программа на C, суммирующая элементы двумерного массива, должна содержать вложенные операторы **for**.

f) Средним, медианой и наиболее вероятным значением для следующего набора значений являются соответственно 5, 6 и 7: 1, 2, 5, 6, 7, 7, 7.

6.8. Напишите операторы C для выполнения каждой из следующих задач:

a) Отобразите на экране значение седьмого элемента символьного массива **f**.

b) Введите значение в элемент номер 4 одномерного массива **b**, состоящего из элементов с плавающей точкой.

c) Инициализируйте каждый из 5 элементов одномерного целочисленного массива **g** числом **8**.

d) Просуммируйте элементы массива **c**, состоящего из 100 элементов с плавающей точкой.

e) Скопируйте массив **a** в начало массива **b**. Предположите, что **float a[11], b[34];**

f) Определите и выведите самое маленькое и самое большое значение, содержащиеся в массиве **w** из 99 элементов с плавающей точкой.

6.9. Рассмотрим целочисленный массив **t** размером 2 на 5.

a) Напишите объявление для **t**.

b) Сколько строк в **t**?

c) Сколько столбцов в **t**?

d) Сколько элементов в **t**?

e) Напишите имена всех элементов второй строки массива **t**.

f) Напишите имена всех элементов третьего столбца массива **t**.

g) Напишите оператор C, который устанавливает в нуль элемент массива **t** в строке 1 и столбце 2.

h) Напишите ряд операторов C, инициализирующих каждый элемент массива **t** нулем. Не используйте структуру повторения.

i) Напишите вложенную структуру **for**, инициализирующую каждый элемент массива **t** нулем.

j) Напишите оператор C для ввода значений в элементы массива **t** с терминала.

k) Напишите ряд операторов C для определения и вывода наименьшего значения в массиве **t**.

l) Напишите оператор C для отображения на экране элементов первой строки массива **t**.

m) Напишите оператор C для суммирования элементов четвертого столбца массива **t**.

n) Напишите ряд операторов C, которые выводят элементы массива **t** в наглядной табличной форме. Перечислите индексы столбцов в качестве заголовков в верхней части таблицы и индексы строк слева от каждой строки.

6.10. Используйте одномерный массив для решения следующей задачи. Компания оплачивает работу своих продавцов с учетом комиссионных. Продавцы получают $200 в неделю плюс 9% от их валового сбыта. Например, продавец, валовой сбыт которого составляет $3000 в неделю, получит $200 плюс 9% от $3000, или в сумме $470. Напишите программу на C (с массивом счетчиков) для определения количества продавцов, заработок которых попал в каждый из следующих диапазонов (предположите, что заработок каждого продавца округлен до целочисленного значения):

1. $201-$299
2. $300-$399
3. $400-$499
4. $500-$599
5. $600-$699
6. $700-$799
7. $800-$899
8. $900-$999
9. $1000 и более

6.11. Метод пузырьковой сортировки, представленный на рис. 6.15, является неэффективным для больших массивов. Сделайте следующие простые изменения для повышения эффективности пузырьковой сортировки.

a) После первого прохода самое большое число гарантированно находится в элементе массива с наибольшим индексом; после второго прохода «на место» попадают два самых больших числа и так далее. Измените пузырьковую сортировку так, чтобы вместо проведения девяти сравнений на каждом проходе нужно было делать восемь сравнений на втором проходе, семь на третьем и т.д.

b) Данные в массиве могут уже находиться в надлежащем порядке или порядке, близком к нему, так зачем делать девять проходов, если будет достаточно меньшего количества? Измените сортировку так, чтобы в конце каждого прохода проверялось, делались ли перестановки. Если перестановок не было, то данные уже находятся в нужном порядке, так что программа должна завершиться. Если перестановки были, то необходим по крайней мере еще один проход.

6.12. Напишите одиночный оператор для выполнения каждой из следующих операций над одномерным массивом:

a) Инициализируйте 10 элементов целочисленного массива **counts** нулями.

b) Прибавьте 1 к каждому из 15 элементов целочисленного массива **bonus**.

c) Считайте с клавиатуры 12 значений массива **monthlyTemperatures**, состоящего из элементов с плавающей точкой.

d) Выведите 5 значений целочисленного массива **bestScores** в виде столбца.

6.13. Найдите ошибку(и) в каждом из следующих операторов:

a) Предположим: **char str[5];**

```
scanf("%s", str); /* Пользователь вводит слово Hello */
```

b) Предположим: **int a[3];**

```
printf ( "$d %d %d\n", a[1], a[2], a[3]);
```

c)
```
float f[3] = {1.1, 10.01, 100.001, 1000.0001};
```

d) Предположим: **double d[2][10];**

```
d[1, 9] = 2.345;
```

6.14 Измените программу на рис. 6.16 так, чтобы функция **mode** могла обрабатывать случай нескольких равных сумм при расчете наиболее вероятного значения. Также измените функцию **median** так, чтобы в массиве с четным числом элементов усреднялись значения двух элементов в середине.

6.15. Используйте одномерный массив для решения следующей задачи. Прочитайте 20 чисел, каждое из которых находится между 10 и 100, включая эти значения. В процессе считывания выводите это число только в том случае, если оно не дублирует одного из уже считанных чисел. Предусмотрите «наихудший случай», когда все 20 чисел различны. Объявите для решения этой задачи массив наименьшего возможного размера.

6.16. Промаркируйте элементы двумерного массива **sales** размером 3 на 5, чтобы показать порядок, в котором они устанавливаются в нуль в следующем фрагменте программы:

```
for (row = 0; row <= 2; row++)
   for (column = 0; column <= 4; column++)
      sales[row][column] = 0;
```

6.17. Что делает следующая программа?

```
#include <stdio.h>
#define SIZE 10

int WhatIsThis(int [], int);

main()
{
   int total, a[SIZE] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
```

```
   total = WhatIsThis(a, SIZE);
   printf("Total of array element values id %d\n", total);
   return 0;
}

int WhatIsThis(int b[], int size)
{
   if (size == 1)
      return b[0];
   else
      return b[size - 1] + WhatIsThis(b, size - 1);
}
```

6.18. Что делает следующая программа?

```
#include <stdio.h>
#define SIZE 10

void someFunction(int [], int);

main()
{
   int a[SIZE] = {32, 27, 64, 18, 95, 14, 90, 70, 60, 37};

   printf("The values in the array are:\n", total);
   someFunction(a, SIZE);
   printf("\n");
   return 0;
}

void someFunction(int b[], int size)
{
   if (size > 0) {
      someFunction(&b[1], size - 1);
      printf("%d ", b[0]);
   }
}
```

6.19. Напишите программу на C, которая моделирует бросание двух игральных костей. Программа должна вызывать функцию **rand** для «бросания» первой кости и затем эту же функцию для «бросания» второй. Затем должна быть вычислена сумма двух значений. Замечание: поскольку на каждой кости может выпадать целое значение от 1 до 6, то сумма двух значений может изменяться от 2 до 12, при этом 7 будет наиболее часто встречающейся суммой, а 2 и 12 — встречающимися наименее часто. На рис. 6.23 показаны 36 возможных комбинаций для двух костей. Ваша программа должна бросить две кости 36,000 раз. Используйте одномерный массив для подсчета числа появлений каждой из возможных сумм. Выведите результаты в табличной форме. Также определите, являются ли итоговые суммы правдоподобными, т.е. поскольку для выбрасывания 7 существует шесть способов, то все выбрасывания 7 должны составить приблизительно одну шестую.

	1	2	3	4	5	6
1	2	3	4	5	6	7
2	3	4	5	6	7	8
3	4	5	6	7	8	9
4	5	6	7	8	9	10
5	6	7	8	9	10	11
6	7	8	9	10	11	12

Рис. 6.23. 36 возможных исходов при бросании двух костей

6.20. Напишите программу, в которой разыгрываются 1000 партий в крепс и которая отвечает на каждый из следующих вопросов:

a) Сколько игр выигрывается при первом бросании, втором бросании, ..., двадцатом бросании и после двадцатого бросания?

b) Сколько игр проигрывается при первом бросании, втором бросании, ..., двадцатом бросании и после двадцатого бросания?

c) Какова вероятность выигрыша в крепс? (Замечание: вы должны знать, что крепс является одной из самых справедливых игр в казино. Как вы полагаете, что это означает?)

d) Какова средняя продолжительность игры в крепс?

e) Увеличиваются ли шансы на выигрыш при большей продолжительности игры?

6.21. (*Система предварительной продажи билетов*) Небольшая авиакомпания недавно приобрела компьютер для новой автоматизированной системы предварительной продажи билетов. Президент компании попросил вас написать на C программное обеспечение для новой системы. Вы должны составить программу для бронирования мест на каждом рейсе единственного самолета авиакомпании (его вместимость: 10 мест).

Ваша программа должна отображать на экране следующее меню альтернативных возможностей:

```
Пожалуйста, введите 1 для "курящих"
Пожалуйста, введите 2 для "некурящих"
```

Если оператор вводит 1, то ваша программа должна забронировать место в отсеке для курящих (места 1–5). Если оператор вводит 2, то ваша программа должна забронировать место в отсеке для некурящих (места 6–10). После этого ваша программа должна напечатать посадочный талон с указанием номера места пассажира и информации о том, находится ли оно в отсеке для курящих или в отсеке для некурящих пассажиров самолета.

Для представления схемы мест для пассажиров самолета используйте одномерный массив. Инициализируйте все элементы массива нулями, чтобы показать, что все места свободны. По мере бронирова-

ния каждого места устанавливайте соответствующие элементы массива в 1 для указания на то, что это место больше не является свободным.

Конечно, ваша программа никогда не должна бронировать уже забронированное место. Если отсек для курящих заполнен, ваша программа должна запросить оператора, допустимо ли бронирование места в отсеке для некурящих (и наоборот). В случае положительного ответа произведите соответствующее бронирование места. В случае отрицательного ответа выведите сообщение **«Следующий рейс через 3 часа.»**

6.22. Используйте двумерный массив для решения следующей задачи. В компании работают четыре продавца (с 1 по 4), которые продают пять различных видов изделий (с 1 по 5). Один раз в день каждый продавец передает в компанию карточку сбыта по каждой разновидности проданного товара. Каждая карточка содержит:

1. Номер продавца
2. Номер изделия
3. Общую сумму в долларах за данный товар, проданный в этот день

Таким образом, каждый продавец передает в день от 0 до 5 карточек сбыта. Предположим, что доступна информация по всем карточкам за последний месяц. Напишите программу, которая будет считывать всю эту информацию о сбыте за последний месяц и подводить общий итог о сбыте каждым продавцом каждой разновидности товара. Все итоговые суммы должны храниться в двумерном массиве **sales**. После обработки всей информации за последний месяц выведите результаты в виде таблицы, в которой каждый столбец представляет конкретного продавца и каждая строка представляет конкретную разновидность товара. Для получения общего сбыта каждой разновидности товара за последний месяц просуммируйте каждую строку; для получения общего сбыта для каждого продавца за последний месяц проведите суммирование каждого столбца. Распечатка вашей таблицы должна включать эти перекрестные итоговые суммы справа от итоговых строк и в нижней части итоговых столбцов.

6.23. (*Черепашья графика*) Язык Лого, особенно популярный среди пользователей персональных компьютеров, сделал знаменитым принцип черепашьей графики (turtle graphics). Вообразите себе механическую черепаху, которая передвигается по комнате под управлением программы на С. Черепаха держит перо в одной из двух позиций, верхней или нижней. Когда перо находится в нижней позиции, черепаха при своем движении вычерчивает различные фигуры, а когда перо находится в верхней позиции, черепаха свободно перемещается, не оставляя при этом никаких следов. При решении этой задачи вы смоделируете действия черепахи, а также создадите автоматизированный блокнот для набросков.

Используйте массив **floor** размером 50 на 50, инициализированный нулями. Считывайте команды из содержащего их массива. Все время отслеживайте текущее местоположение черепахи и то, находится ли перо в настоящее время в верхней или нижней позиции.

Предположим, что черепаха всегда начинает в точке пола с координатами 0,0 с поднятым пером. Множество команд черепахи, которые должна обрабатывать ваша программа, таково:

Команда	Смысл
1	Перо вверх
2	Перо вниз
3	Повернуть направо
4	Повернуть налево
5, 10	Переместиться вперед на 10 (или другое число) интервалов
6	Вывести на экран массив размером 50x50
9	Конец данных (контрольное значение)

Предположим, что черепаха находится где-то вблизи от центра пола. Следующая «программа» рисует и выводит на экран квадрат размером 12 x 12:

```
2
5,12
3
5,12
3
5,12
3
5,12
1
6
9
```

Когда черепаха перемещается с опущенным пером, устанавливайте соответствующие элементы массива **floor** в **1**. При выдаче команды **6** (вывести на экран) для всех элементов массива, в которых находится **1**, отобразите на экране звездочку или любой другой символ, который вам нравится. Для всех элементов массива, в которых находится нуль, отобразите на экране пробел. Напишите программу на C, реализующую возможности черепашьей графики, которые мы сейчас обсудили. Напишите несколько программ черепашьей графики, которые рисуют интересные фигуры. Добавьте другие команды для увеличения мощности вашего «черепашьего» языка.

6.24. (*Обход конем*) Одной из наиболее интересных головоломок для любителей шахмат является задача об обходе конем шахматной доски, первоначально предложенная математиком Эйлером. Проблема состоит в следующем: может ли шахматная фигура, называемая конем, обойти пустую шахматную доску, при этом побывав в каждой из 64 клеток один и только один раз? Здесь мы глубоко исследуем эту захватывающую проблему.

Конь ходит в виде буквы L (на две клетки в одном направлении и затем на одну клетку в перпендикулярном ему). Таким образом, из квадрата в центре пустой шахматной доски конь может сделать восемь различных ходов, показанных на рис. 6.24 (они пронумерованы от 0 до 7).

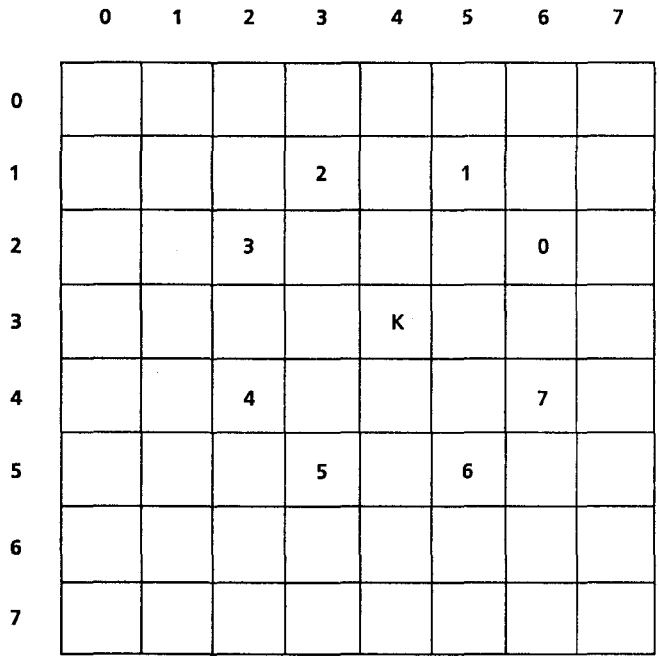


Рис. 6.24. Восемь возможных ходов конем

a) Нарисуйте на листе бумаги шахматную доску размером 8 на 8 и попытайтесь обойти ее конем самостоятельно. Поместите **1** в первую клетку, в которую вы перемещаетесь, **2** во вторую клетку, **3** в третью и т.д. Перед началом обхода оцените, насколько далеко, по вашему мнению, вы сможете продвинуться, помня о том, что полностью обход доски состоит из 64 ходов. Как далеко вы продвинулись? Близко ли вы подошли к своей оценке?

b) Теперь давайте разработаем программу, которая будет перемещать коня по шахматной доске. Сама доска представлена двумерным массивом **board** размером 8 на 8. Каждая клетка инициализирована нулем. Мы описываем каждый из восьми возможных ходов его горизонтальной и вертикальной составляющими. Например, ход типа 0, как показано на рис. 6.24, состоит из перемещения на две клетки вправо по горизонтали и на одну клетку вверх по вертикали. Ход 2 состоит из перемещения на одну клетку влево по горизонтали и на две клетки вверх по вертикали. Горизонтальные перемещения влево и вертикальные перемещения вверх обозначены отрицательными числами. Указанные восемь ходов могут быть описаны с помощью двух одномерных массивов, **horizontal** и **vertical**, следующим образом:

```
horizontal[0] = 2
horizontal[1] = 1
horizontal[2] = -1
horizontal[3] = -2
horizontal[4] = -2
horizontal[5] = -1
horizontal[6] = 1
horizontal[7] = 2

vertical[0] = -1
vertical[1] = -2
vertical[2] = -2
vertical[3] = -1
vertical[4] = 1
vertical[5] = 2
vertical[6] = 2
vertical[7] = 1
```

Пусть переменные **currentRow** и **currentColumn** указывают на строку и столбец текущего местоположения коня. Чтобы сделать ход типа **moveNumber**, где значение **moveNumber** находится между 0 и 7, ваша программа использует операторы

```
currentRow += vertical[moveNumber];
currentColumn += horizontal[moveNumber];
```

Организуйте счетчик, который изменяется от **1** до **64**. Записывайте последнее значение счетчика в каждой клетке, в которую перемещается конь. Не забывайте проверять каждый возможный ход, чтобы определить, не был ли уже конь в этой клетке. И, конечно, проверяйте каждый возможный ход, чтобы убедиться, что конь не пойдет за пределы доски. А теперь напишите программу для перемещения коня по шахматной доске. Выполните программу. Сколько ходов сделал конь?

c) Пытаясь написать и выполнить программу обхода конем шахматной доски, вы, по всей вероятности, развили в себе некие ценные интуитивные качества. Мы воспользуемся ими для разработки *эвристики* (или стратегии) для перемещений коня. Эвристики не гарантируют успеха, однако тщательно разработанная эвристика значительно увеличивает шансы на успех. Вы, возможно, уже заметили, что внешние клетки в некотором смысле вызывают большие проблемы, чем клетки, находящиеся ближе к центру доски. На самом деле, наиболее трудными, или недоступными, являются четыре угловых клетки.

Интуиция подсказывает, что сначала следует попытаться помещать коня в наиболее труднодоступные клетки и оставить напоследок те из них, попасть в которые наиболее просто, с тем, чтобы имелся больший шанс на успех, когда ближе к концу обхода доска будет переполнена.

Мы можем разработать «эвристику доступности», классифицировав каждую из клеток в соответствии с тем, насколько доступной она является, и в дальнейшем всегда перемещать коня в клетку (разумеется, в пределах L-образных ходов коня), которая является наименее доступной. Мы размечаем двумерный массив **accessibility** числа-

ми, указывающими число клеток, из которых каждая конкретная клетка является доступной. Для пустой шахматной доски центральные клетки, следовательно, оцениваются **8**-ю, угловые клетки оцениваются **2**-мя и другие клетки имеют показатели доступности **3**, **4** или **6**, как указано ниже:

```
2 3 4 4 4 4 3 2
3 4 6 6 6 6 4 3
4 6 8 8 8 8 6 4
4 6 8 8 8 8 6 4
4 6 8 8 8 8 6 4
4 6 8 8 8 8 6 4
3 4 6 6 6 6 4 3
2 3 4 4 4 4 3 2
```

Теперь напишите вариант программы обхода конем шахматной доски с использованием эвристики доступности. В любой момент конь должен перемещаться в клетку с наименьшим показателем доступности. В случае нескольких клеток с равными показателями конь может перемещаться в любую из этих клеток. Таким образом, обход может начинаться в любой из четырех угловых клеток. (Замечание: по мере перемещения коня по шахматной доске ваша программа должна уменьшать показатели доступности, поскольку все больше и больше клеток становятся занятыми. В этом случае в любой данный момент времени в процессе обхода показатель доступности каждой доступной клетки остается равным в точности числу клеток, из которых до нее можно добраться.) Выполните эту версию своей программы. Получился ли у вас полный обход? Теперь измените программу для выполнения 64 обходов, по одному из каждой клетки шахматной доски. Сколько у вас получилось полных обходов?

d) Напишите версию программы обхода конем шахматной доски, которая, встретившись с равными показателями для двух или более клеток, решает, какую клетку выбрать на основании результатов просмотра тех клеток, до которых можно добраться из клеток с равными показателями. Ваша программа должна перемещаться в ту клетку, следующее перемещение из которой пришлось бы на клетку с наименьшим показателем доступности.

6.25. (*Обход конем: подходы «с позиции грубой силы»*) В упражнении 6.24 мы нашли решение проблемы, связанной с обходом конем шахматной доски. Использованный подход, названный нами «эвристикой доступности», порождает множество решений и эффективно выполняется.

Поскольку мощность компьютеров продолжает возрастать, мы сможем решать многие проблемы исключительно за счет производительности компьютеров, применяя относительно несложные алгоритмы. Давайте назовем этот подход решением задачи «с позиции грубой силы».

a) Используйте генерацию случайных чисел для предоставления коню возможности обхода шахматной доски (разумеется, в рамках его правильных L-образных ходов) в произвольном порядке. Ваша

программа должна выполнить один обход и вывести конечное состояние шахматной доски. Как далеко забрался ваш конь?

b) По всей вероятности, результатом предыдущей программы явился относительно короткий обход. Теперь измените вашу программу, чтобы она 1000 раз попыталась сделать обход. Объявите одномерный массив для отслеживания числа обходов каждой продолжительности. После того, как ваша программа закончит 1000 попыток обходов, она должна вывести эту информацию в удобной табличной форме. Каков был наилучший результат?

c) По всей вероятности, в результате работы предыдущей программы у вас появились какие-то «приличные» обходы, но не было полных. Теперь «снимите все ограничители» и просто дайте вашей программе выполняться до тех пор, пока она не выдаст полного обхода. (Предостережение: эта версия программы может часами выполняться на мощном компьютере.) Снова организуйте таблицу числа обходов каждой продолжительности и выведите эту таблицу после нахождения первого полного обхода. Сколько обходов попыталась сделать ваша программа перед тем, как был выдан полный обход? Сколько для этого потребовалось времени?

d) Сравните решение задачи об обходе конем шахматной доски «с позиции грубой силы» с вариантом, основанным на эвристике доступности. Какое решение требует более тщательного анализа проблемы? Какой алгоритм было труднее разработать? Для какого из них требуется большая производительность компьютера? Можем ли мы быть уверенными (заранее) в получении полного обхода при использовании эвристики доступности? Можем ли мы быть уверенными (заранее) в получении полного обхода при использовании подхода «с позиции грубой силы»? Обсудите все за и против решения задач «с позиции грубой силы» в общем случае.

6.26. (*Восемь ферзей*) Другой головоломкой для любителей шахмат является задача о восьми ферзях. В простой постановке она выглядит так: можно ли разместить восемь ферзей на пустой шахматной доске таким образом, чтобы ни один из них не «нападал» на любого другого? Это значит, что никакие два ферзя не расположены в одном и том же горизонтальном ряду, в одном и том же вертикальном ряду или на одной диагонали. Используйте подход, применявшийся в упражнении 6.24, чтобы сформулировать эвристику для решения задачи о восьми ферзях. Выполните свою программу. (Подсказка: можно присвоить каждой клетке шахматной доски числовое значение, показывающее, сколько клеток пустой шахматной доски «исключаются из рассмотрения» при помещении ферзя в эту клетку. Например, каждой из четырех угловых клеток было бы присвоено значение 22, как показано на рис. 6.25.)

Если разместить эти «показатели исключений» во всех 64 клетках доски, соответствующей эвристикой могла бы стать следующая: поместить следующего ферзя в клетку с наименьшим показателем исключений. Почему эта стратегия является интуитивно притягательной?

```
********
**
* *
*  *
*   *
*    *
*     *
*      *
*       *
```

Рис. 6.25. 22 клетки, исключаемые при помещении ферзя в верхний левый угол

6.27. (*Восемь ферзей: подходы «с позиции грубой силы»*) В этой задаче вы разработаете несколько подходов «с позиции грубой силы» к решению задачи о восьми ферзях, представленной в упражнении 6.26.

a) Решите задачу о восьми ферзях, используя основанный на случайных числах метод решения «с позиции грубой силы», применявшийся в упражнении 6.25.

b) Используйте метод прямого перебора, т.е. попробуйте все возможные комбинации расположения восьми ферзей на шахматной доске.

c) Как вы думаете, почему подход «с позиции грубой силы», основанный на переборе всех возможностей, может оказаться неподходящим для решения задачи об обходе конем шахматной доски?

d) Сравните и сопоставьте в общем случае подход «с позиции грубой силы», основанный на случайных числах, с подходом, основанным на переборе всех возможностей.

6.28. (*Исключение дублирующих значений*) В главе 12 мы изучим структуру данных двоичного дерева поиска. Одной из особенностей двоичного дерева поиска является то, что при проведении вставок в это дерево дублирующие значения отбрасываются. Это называется исключением дублирующих значений. Напишите программу, которая выдает 20 случайных чисел между 1 и 20. Программа должна сохранять все недублированные значения в массиве. Используйте для решения этой задачи наименьший из возможных массивов.

6.29. (*Обход конем: случай замкнутого обхода*) В задаче об обходе конем шахматной доски обход считается полным, если конь делает 64 хода, побывав в каждой клетке шахматной доски один и только один раз. Замкнутый обход возникает в том случае, если 64-е перемещение удалено на один ход от того места, с которого конь начал обход. Измените программу обхода конем шахматной доски, написанную вами в упражнении 6.24, для проверки наличия замкнутого обхода в том случае, если имеет место полный обход.

6.30. (*Решето Эратосфена*) Простым называется любое целое число, которое можно разделить без остатка только на него само и на 1. Решето Эратосфена представляет собой метод нахождения простых чисел. Оно работает следующим образом:

1) Создайте массив и инициализируйте все его элементы единицей (TRUE). Элементы массива с простыми индексами останутся равными 1. Все другие элементы массива будут по завершении алгоритма установлены в нуль.

2) Начните с индекса массива, равного 2 (индекс 1 должен быть простым); всякий раз при нахождении элемента массива со значением 1 выполните цикл по оставшимся элементам массива и установите в нуль те из них, индекс которых является кратным индексу элемента со значением 1. Для индекса массива, равного 2, все элементы массива после 2, которые кратны 2, будут установлены в нуль (индексы 4, 6, 8, 10 и т.д.). Для индекса массива, равного 3, все элементы массива после 3, кратные 3, будут установлены в нуль (индексы 6, 9, 12, 15 и т.д.).

После завершения этого процесса те элементы массива, которые все еще установлены в единицу, указывают на то, что их индекс является простым числом. Эти индексы могут быть затем выведены на печать. Напишите программу, которая объявляет массив из 1000 элементов для определения и вывода на печать простых чисел между 1 и 999. Исключите из рассмотрения элемент массива с номером 0.

6.31. (*Блочная сортировка*) Для блочной сортировки требуется одномерный массив положительных целых чисел, подлежащий сортировке, и двумерный массив целых чисел с индексами строк от 0 до 9 и индексами столбцов от 0 до n - 1, где n — число значений в сортируемом массиве. Каждая строка двумерного массива называется блоком. Напишите функцию **bucketSort**, которая принимает в качестве параметров целочисленный массив и его размер.

Алгоритм выглядит так:

1) Выполните цикл по одномерному массиву и поместите каждое из его значений в некоторую строку блочного массива в зависимости от значения его разряда, представляющего единицы. Например, 97 помещается в строку 7, 3 помещается в строку 3 и 100 помещается в строку 0.

2) Выполните цикл по блочному массиву и скопируйте его значения обратно в исходный массив. Новый порядок для приведенных выше значений в одномерном массиве будет 100, 3 и 97.

3) Повторите этот процесс для каждого из последующих разрядов (десятков, сотен, тысяч и т.д.) и прекратите его выполнение после обработки крайнего слева разряда самого большого числа.

При втором проходе по массиву 100 помещается в строку 0, 3 помещается в строку 0 (у 3 только один разряд) и 97 помещается в строку 9. Порядок значений в одномерном массиве будет 100, 3 и 97. При третьем проходе 100 помещается в строку 1, 3 помещается в строку нуль и 97 помещается в строку нуль (после 3). Гарантируется, что при блочной сортировке все значения будут надлежащим образом отсортированы после обработки крайнего слева разряда самого большого числа. Блочная сортировка заканчивается, когда все значения будут скопированы в строку двумерного массива с номером 0.

Обратите внимание, что размер двумерного массива, содержащего блоки, в десять раз превосходит размер целочисленного массива, подлежащего сортировке. Этот метод сортировки обеспечивает большую эффективность в сравнении с пузырьковым методом, но требует значительно большего объема памяти. При пузырьковой сорти-

ровке требуется только одна дополнительная ячейка памяти для сортируемого типа данных. Блочная сортировка является примером уступки в отношении требуемой памяти ради уменьшения времени выполнения. Она использует больший объем памяти, но выполняется быстрее. Данный вариант блочной сортировки требует на каждом шаге копирования всех данных обратно в исходный массив. Другая возможность состоит в создании второго двумерного блочного массива и многократном перемещении данных между этими двумя блочными массивами до тех пор, пока все данные не будут скопированы в строку с номером нуль одного из массивов. Тогда строка с номером нуль будет содержать отсортированный массив.

Упражнения на рекурсию

6.32. (*Выборочная сортировка*) При выборочной сортировке происходит поиск наименьшего элемента в массиве. Когда наименьший элемент найден, его меняют местами с первым элементом массива. Затем процесс повторяется для подмассива, начинающегося со второго элемента массива. Каждый проход по массиву приводит к помещению одного элемента на подходящее для него место. Эта сортировка при обработке данных имеет характеристики, сходные с пузырьковым методом — по массиву из n элементов должно быть сделано n - 1 проходов и в каждом подмассиве для нахождения наименьшего значения должно быть сделано n - 1 сравнений. Если обрабатываемый подмассив содержит один элемент, массив отсортирован. Напишите рекурсивную функцию **selectionSort** для выполнения этого алгоритма.

6.33. (*Палиндромы*) Палиндромом называется строка, которая читается одинаково как слева направо, так и в обратном направлении. Некоторыми примерами палиндромов являются: «radar», «able was i ere i saw elba» и «a man a plan a canal Panama». Напишите рекурсивную функцию **testPalindrome**, которая возвращает 1, если строка, хранящаяся в массиве, является палиндромом, и 0 в противном случае. Функция должна игнорировать пробелы и знаки пунктуации.

6.34. (*Линейный поиск*) Модифицируйте программу на рис. 6.18, чтобы для выполнения последовательного перебора массива можно было использовать рекурсивную функцию **linearSearch**. Эта функция должна принимать в качестве параметров целочисленный массив и его размер. Если ключ поиска найден, возвратите индекс массива; в противном случае возвратите -1.

6.35. (*Двоичный поиск*) Модифицируйте программу на рис. 6.19, чтобы для выполнения двоичного поиска в массиве можно было использовать рекурсивную функцию **binarySearch**. Эта функция должна принимать в качестве параметров целочисленный массив и начальный и конечный индексы. Если ключ поиска найден, возвратите индекс массива; в противном случае возвратите -1.

6.36. (*Восемь ферзей*) Модифицируйте программу о восьми ферзях, созданную вами в упражнении 6.26, для рекурсивного решения задачи.

6.37. (*Вывод массива*) Напишите рекурсивную функцию **printArray**, которая принимает в качестве параметров целочисленный массив и его размер и ничего не возвращает. Функция должна прекратить обработку и завершить свое выполнение при получении массива нулевого размера.

6.38. (*Вывод строки в обратном порядке*) Напишите рекурсивную функцию **stringReverse**, которая принимает в качестве параметра символьный массив и ничего не возвращает. Функция должна прекратить обработку и завершить свое выполнение, если будет встречен завершающий строку нулевой символ.

6.39. (*Нахождение минимального значения в массиве*) Напишите рекурсивную функцию **recursiveMinimum**, которая принимает в качестве параметров целочисленный массив и его размер и возвращает наименьший элемент массива. Функция должна прекратить обработку и завершить свое выполнение при получении массива из 1 элемента.

Указатели



Цели

- Понять концепцию указателей.
- Изучить применение указателей для передачи аргументов в вызове функции по ссылке.
- Понять связь между указателями, массивами и строками.
- Познакомиться с использованием указателей на функции.
- Научиться объявлять и использовать массивы строк.

Содержание

7.1. Введение
7.2. Объявление и инициализация переменной-указателя
7.3. Операции над указателями
7.4. Передача параметра по ссылке
7.5. Использование модификатора const с указателями
7.6. Программа пузырьковой сортировки, использующая вызов по ссылке
7.7. Выражения и арифметические операции с указателями
7.8. Связь между указателями и массивами
7.9. Массивы указателей
7.10. Пример: программа тасовки и сдачи колоды карт
7.11. Указатели на функции

Резюме • *Распространенные ошибки программирования* • *Хороший стиль программирования* • *Советы по повышению эффективности* • *Советы по переносимости программ* • *Общие методические замечания* • *Упражнения для самоконтроля* • *Ответы на упражнения для самоконтроля* • *Упражнения* • *Специальный раздел: как самому построить компьютер*

7.1. Введение

В этой главе мы обсудим один из наиболее мощных элементов языка C — *указатели*. Овладение техникой работы с указателями — дело достаточно трудное. Указатели дают возможность программам генерировать вызов по ссылке, создавать и управлять динамическими структурами данных, которые могут увеличиваться и уменьшаться в размере, — это могут быть, например, связанные списки, очереди, стеки и деревья. Эта глава посвящена основам работы с указателями. В главе 10 будет рассказано о роли указателей в работе со структурами. В главе 12 будут представлены динамические методы управления памятью с примерами создания и использования динамических структур данных.

7.2. Объявление и инициализация переменной-указателя

Указатели представляют собой переменные, значениями которых являются адреса памяти. В «обычной» переменной непосредственно содержится некоторое значение. Указатель же содержит адрес переменной, в которой находится конкретное значение. Говорят, что переменная *непосредственно* ссылается на значение, а указатель *косвенно* ссылается на значение (рис. 7.1). Ссылка на значение через посредство указателя называется *косвенной адресацией*.

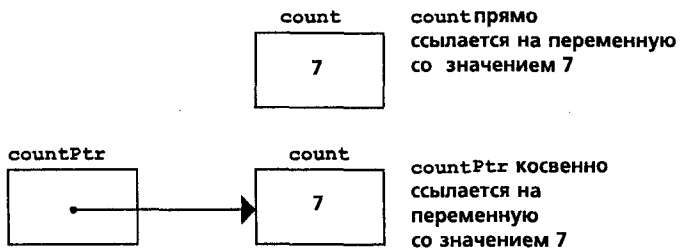


Рис. 7.1. *Прямая и косвенная ссылка на переменную*

Указатели, как и любые другие переменные, должны быть объявлены, прежде чем они будут использоваться. В операторе

```
int *countPtr, count;
```

объявляется переменная **countPtr** типа **int *** (указатель на целочисленное значение), что читается как «**countPtr** — указатель на целое» или «**countPtr** указывает на объект типа целое». Кроме того, объявляется переменная **count** целого типа. Символ * в объявлении распространяется только на **countPtr**. Этот символ означает, что объявляемая переменная является указателем. Можно объявлять указатели, ссылающиеся на объекты любого типа.

Распространенная ошибка программирования 7.1

Знак операции косвенной адресации * не распространяется на все переменные в строке объявления. Символ * должен предшествовать имени каждого указателя.

Хороший стиль программирования 7.1

Включайте в имена переменных-указателей префикс (или суффикс) **ptr**; такие имена подскажут вам, что эти переменные являются указателями и должны обрабатываться соответствующим образом.

Указатели должны быть инициализированы либо при объявлении, либо при помощи оператора присваивания. Указатель может быть инициализирован нулем, макросом **NULL** или значением адреса. Указатель со значением **NULL** не указывает ни на что. Символическая константа **NULL** определяется в файле заголовка **<stdio.h>** (и в некоторых других заголовочных файлах). Инициализация указателя значением **0** эквивалентна инициализации указателя константой **NULL**, однако использование **NULL** предпочтительнее. Когда присваивается значение **0**, то происходит его преобразование к указателю

соответствующего типа. Значение **0** является единственным целым числом, которое может быть присвоено переменной-указателю непосредственно. Присвоение указателю адреса обсуждается в разделе 7.3.

Хороший стиль программирования 7.2

Чтобы избежать неожиданных результатов, всегда инициализируйте указатели.

7.3. Операции над указателями

Операция *взятия адреса* **&** является унарной операцией, которая возвращает адрес своего операнда. Например, если мы объявим переменные

```
int y = 5;
int * yPtr;
```

то следующий оператор

```
yPtr = &y;
```

присвоит переменной-указателю **yPtr** адрес переменной **y**. После этого можно говорить, что переменная **yPtr** «указывает на» **y**. На рис. 7.2 схематически представлены значения переменных после выполнения приведенного оператора присваивания.

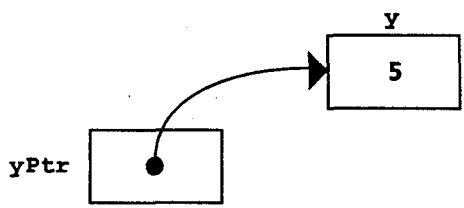


Рис. 7.2. Графическое представление указателя на переменную целого типа

На рис. 7.3 показано расположение переменных в памяти, в предположении, что целая переменная **y** находится по адресу **600000**, а переменная-указатель **yPtr** — по адресу **500000**. Операндом операции взятия адреса должна быть переменная; эта операция не может применяться к константам, выражениям или к переменным, объявленным с модификатором **register**.



Рис. 7.3. Размещение и содержимое **y** и **yPtr** в памяти

Операция *****, обычно называемая операцией *косвенной адресации* или *разыменования*, возвращает значение объекта, на который операнд (то есть указатель) ссылается. Например, оператор

```
printf ("%d", *yPtr);
```

выводит значение переменной **y**, а именно число 5. Такое использование операции ***** называется *разыменованием* указателя.

Распространенная ошибка программирования 7.2

Разыменование указателя, который не был инициализирован или которому не присвоили необходимое значение адреса. Приводит к фатальной ошибке во время выполнения программы или случайной порче данных, в результате чего выполнение программы завершается с неверным результатом.

В программе на рис. 7.4 показано выполнение операций над указателями. Спецификатор формата **%p** функции **printf** выводит содержимое элемента памяти как шестнадцатеричное целое число (подробнее относительно шестнадцатеричных целых чисел см. приложение Д, «Системы счисления»). Обратите внимание на то, что адрес переменной **a** и значение **aPtr** при выводе совпадают, из чего можно заключить, что адрес переменной **a** действительно присвоен переменной-указателю **aPtr**. Операции **&** и ***** взаимно дополняют друг друга, и когда они обе применяются последовательно к **aPtr**, то порядок их следования роли не играет — результат будет тем же. В таблице на рис. 7.5 показаны известные нам на данный момент операции, их приоритет и ассоциативность.

```
/* Использование операций & и * */
#include <stdio.h>

main()
{
   int a;      /* a - переменная целого типа */
   int *aPtr; /* aPtr - указатель на целое */

   a = 7;
   aPtr = &a; /* Указателю aPtr присваивается адрес переменной a */

   printf("The address of a is %p\n"
          "The value of aPtr is %p\n\n", &a, aPtr);

   printf("The value of a is %d\n"
          "The value of *aPtr is %d\n\n", a, *aPtr);

   printf("Proving that * and & are complements of each oter. "
          "\n&*aPtr = %p\n*&aPtr = %p\n",
          &*aPtr, *&aPtr);
   return 0;
}
```

```
The address of a is FFF4
The value of aPtr is FFF4

The value of a is 7
The value of *aPtr is 7

Proving that * and & are complements of each oter.
&*aPtr = FFF4
*&aPtr = FFF4
```

Рис. 7.4. Операции **&** и ***** над указателем

Операции	Ассоциативность	Описание
() []	слева направо	высший приоритет
+ - ++ -- ! * & (тип)	справа налево	унарные
* / %	слева направо	мультипликативные
+ -	слева направо	аддитивные
< <= > >=	слева направо	отношения
== !=	слева направо	сравнения
&&	слева направо	логическое И
||	слева направо	логическое ИЛИ
?:	справа налево	условная
= += -= *= /= %=	справа налево	присваивания
,	слева направо	запятая

Рис. 7.5. Старшинство операций

7.4. Передача параметра по ссылке

Существуют два способа передачи параметров функции — по значению и по ссылке. Все вызовы функций в C являются вызовами по значению. Как мы видели в главе 5, при помощи оператора **return** можно возвратить единственное значение из вызываемой функции в вызывающую (или вернуть управление в вызывающую функцию без передачи значения). Однако во многих случаях нужно иметь возможность изменять не одну, а несколько переменных в вызывающей функции, или передавать функции указатель на большой объект данных, чтобы избежать издержек, связанных с передачей параметра по значению (так как в этом случае создается и потом передается копия объекта). Для этих целей в C существует возможность вызова функции по ссылке. В C для организации вызова по ссылке программисты используют указатели и операцию косвенной адресации. Если вызывается функция, аргументы которой должны изменяться, то в этом случае ей передаются адреса аргументов. Обычно для этой цели применяется операция взятия адреса (&) к переменной, значение которой будет изменяться. Как мы уже видели в главе 6, если в качестве аргумента в функцию передается массив, то функция получает адрес начала массива и использовать операцию & в этом случае не нужно (использование имени массива эквивалентно заданию в качестве аргумента **&arrayName[0]**). Когда адрес переменной передан функции, то для изменения ее значения может быть использована операция косвенной адресации (*).

Программы на рис. 7.6 и рис. 7.7 представляют собой два варианта функции, вычисляющей куб целого числа — **cubeByValue** и **cubeByReference**. Программа на рис. 7.6 передает переменную **number** функции **cubeByValue** по значению. В функции **cubeByValue** вычисляется третья степень аргумента функции, и полученное значение возвращается в функцию **main** при помощи оператора **return**. Затем это новое значение присваивается переменной **number** в функции **main**.

```
/ * Вычисление куба числа с использованием вызова по значению * /
#include <stdio.h>

int cubeByValue (int);

main()
{
   int number = 5;

   printf("The original value of number is %d\n", number);
   number = cubeByValue(number);
   printf("The new value of number is %d\n", number);
   return 0;
}

int cubeByValue(int n)
{
   return n * n * n;  /* возводим в куб локальную переменную n */
}
```

```
The original value of number is 5
The new value of number is 125
```

Рис. 7.6. Вычисление куба переменной с вызовом по значению

```
/* Вычисление куба переменной при помощи вызова по ссылке */
#include <stdio.h>

void cubeByReference(int *);

main()
{
   int number = 5;
   printf ("The original value of number is %d\n", number);
   cubeByReference(&number);
   printf("The new value of number is %d\n", number);
   return 0;
}

void cubeByReference(int *nPtr)
{
   *nPtr = *nPtr * *nPtr * *nPtr; /* вычисление куба переменной
                                     number функции main */
}
```

```
The original value of number is 5
The new value of number is 125
```

Рис. 7.7. Вычисление куба переменной с вызовом по ссылке

В программе на рис. 7.7 переменная **number** передается по ссылке, т.е. функция **cubeByReference** получает адрес **number**. Функция **cubeByReference** помещает этот адрес в свой параметр **nPtr**, переменную-указатель на целое. Далее функция разыменовывает указатель и вычисляет кубическое значение числа, на которое указывает **nPtr**. В результате значение переменной number в функции **main** изменяется. На рис. 7.8 и 7.9 графически изображены этапы выполнения программ, приведенных соответственно на рис. 7.6 и рис. 7.7.

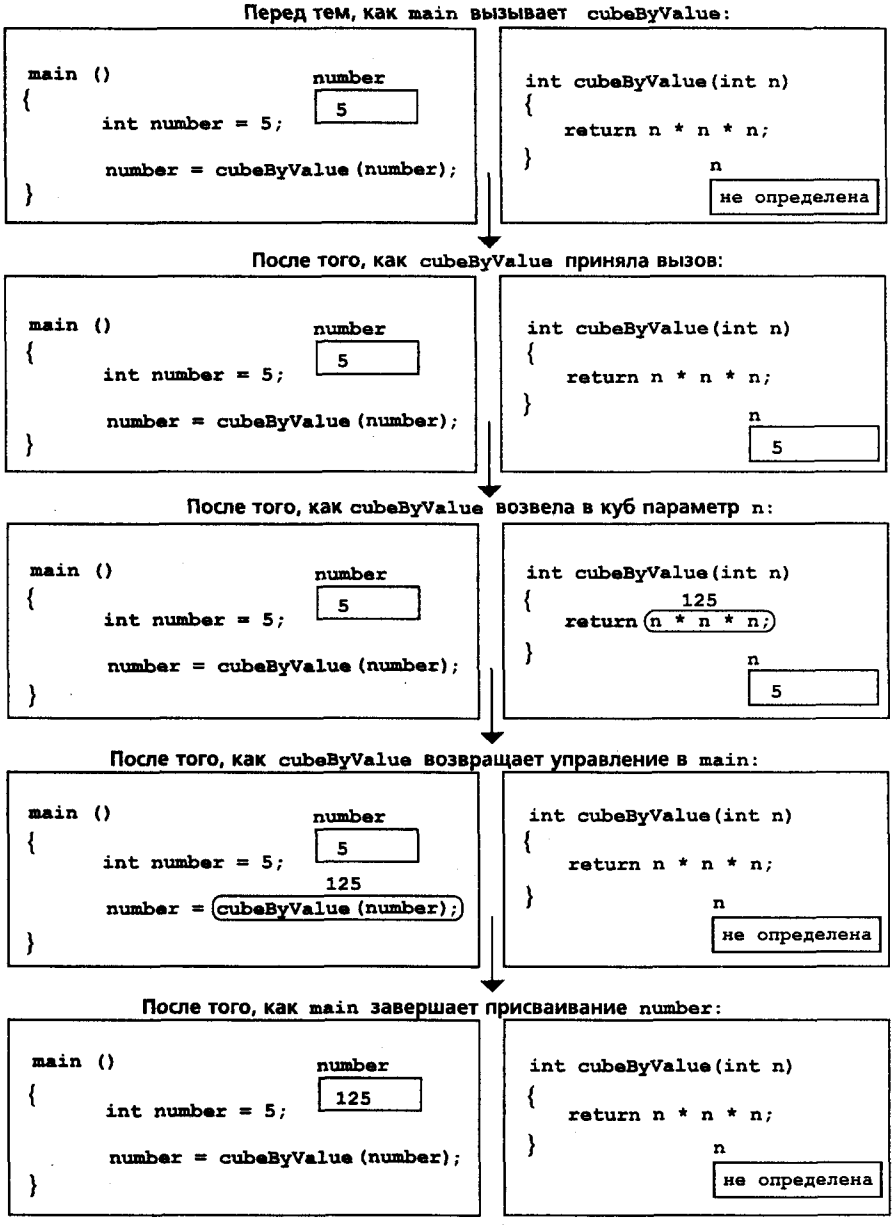


Рис. 7.8. Последовательность выполнения обычного вызова по значению

Распространенная ошибка программирования 7.3

Не разыменовывайте указатель, кроме случая, когда вам нужно получить значение, на которое он ссылается.

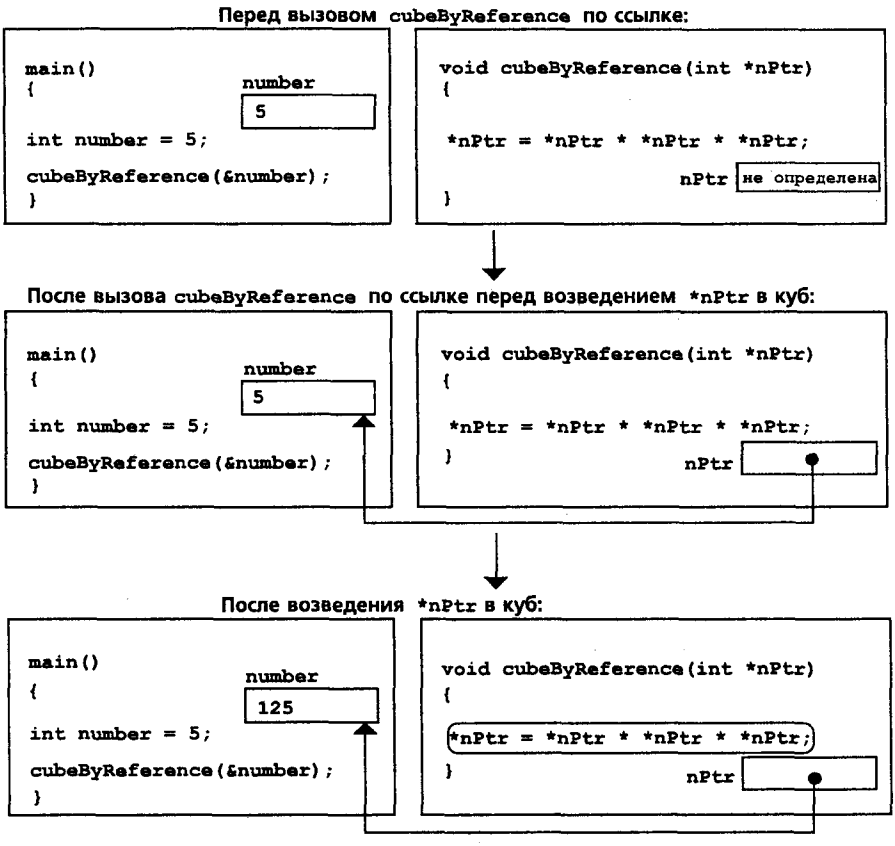


Рис. 7.9. Анализа типичного вызова по ссылке

Функция, получающая в качестве аргумента адрес, должна иметь соответствующий параметр-указатель для размещения адреса. Например, заголовок функции **cubeByReference** имеет вид

```
void cubeByReference(int *nPtr)
```

Заголовок функции показывает, что **cubeByReference** получает в качестве аргумента адрес целой переменной, помещает его в локальную переменную **nPtr** и не возвращает значения.

Прототип функции **cubeByReference** содержит в круглых скобках **int ***. Включать имена указателей, как и имена переменных других типов, в прототипы функции необязательно. Имена включаются как комментарии и игнорируются компилятором.

Если функция может получить в качестве аргумента одномерный массив, то в функциональном заголовке и в прототипе функции соответствующий параметр может быть определен как указатель. Компилятор не делает различия между функцией, имеющей параметром указатель, и функцией, которая получает в качестве аргумента одномерный массив. Однако понятно, что функция должна «знать», когда она получает массив и когда ссылку на одиночную переменную. Когда компилятор встречает одномерный массив в качестве параметра функции, заданный в форме **int b[]**, компилятор преобразует этот параметр к виду **int *b**. Эти две формы являются взаимозаменяемыми.

Хороший стиль программирования 7.3

Используйте вызов по ссылке только в том случае, когда вызываемая функция должна изменять передаваемый ей аргумент. (Еще один пример принципа минимума привилегий.)

7.5. Использование модификатора const с указателями

Модификатор **const** дает возможность программисту сообщить компилятору о том, что значение указанной переменной не должно изменяться. В первых версиях C модификатор **const** отсутствовал, он был добавлен к языку C комитетом ANSI.

Общее методическое замечание 7.1

Употребление модификатора **const** можно рассматривать в рамках соблюдения принципа минимальных привилегий. Следование этому принципу позволяет существенно сократить время отладки программы и количество побочных нежелательных эффектов, а также делает программу проще для понимания и сопровождения.

Совет по переносимости программ 7.1

Хотя модификатор **const** определен в стандарте ANSI C, некоторые системы его не поддерживают.

За прошедшие годы большое количество программ было написано на ранних версиях языка C, в которых не использовался модификатор **const**, потому что его просто не существовало. По этой причине имеются огромные возможности для усовершенствования программного обеспечения, написанного на «старом» C. Хотя и сейчас еще многие программисты, работающие с ANSI C, не используют модификатор **const** в своих программах, потому что они привыкли к старому варианту языка C, и игнорируют многие другие возможности хорошего стиля в разработке программ.

Существует шесть вариантов использования (или неиспользования) модификатора **const** с параметрами функции — две при передаче параметра по значению и четыре при передаче параметра по ссылке. Как выбрать один вариант из шести? Руководствуйтесь принципом минимальных привилегий. Всегда предоставляйте функции столько прав доступа к данным, передаваемым через параметры, сколько нужно для того, чтобы функция выполнила свою задачу, но не более того.

Из главы 5 мы знаем, что все вызовы в C — это вызовы по значению; при вызове функции ей передается копия аргумента. Если копия изменяется в функции, первоначальное значение аргумента в вызвавшей функции остается без изменения. Во многих случаях переданное значение изменяется в процессе выполнения функцией своей задачи. Однако в некоторых случаях переданное значение не должно изменяться, несмотря на то, что вызываемая функция оперирует всего лишь копией первоначального значения.

Рассмотрим функцию, которая принимает одномерный массив и размер массива как параметры и выводит содержимое этого массива. Такая функция

должна содержать цикл по элементам массива и выводить каждый элемент массива в цикле. Размер массива используется для того, чтобы определить последний элемент массива, завершить цикл и вывод элементов массива. Размер массива не должен изменяться в теле функции.

Общее методическое замечание 7.2

Если передаваемое функции значение не изменяется (или не должно быть изменено) в теле функции, оно должно объявляться с модификатором **const**, чтобы гарантировать невозможность даже случайного изменения.

Если происходит попытка изменить значение, которое было объявлено с модификатором **const**, компилятор перехватывает эту попытку и выдает либо предупреждение, либо сообщение об ошибке, в зависимости от того, какой компилятор вы используете.

Общее методическое замечание 7.3

В результате вызова по значению в вызывающей функции может быть изменена только одна величина. Этой величине должно быть присвоено возвращаемое вызванной функцией значение. Для изменения сразу нескольких значений в вызывающей функции необходимо использовать вызов по ссылке.

Хороший стиль программирования 7.4

Прежде чем использовать функцию, посмотрите на ее прототип, чтобы узнать, может ли функция изменять переданные ей значения.

Распространенная ошибка программирования 7.4

Функции ошибочно передается параметр по значению, а не по ссылке. В таких случаях некоторые компиляторы поступят с полученным аргументом как с указателем и произведут разыменование этого «указателя». В результате во время выполнения программы произойдет ошибка обращения к памяти или ошибка сегментации. Компиляторы, проверяющие аргументы и параметры на соответствие типов, выдадут сообщение об ошибке.

Существует четыре способа передать функции указатель: изменяемый указатель на изменяемые данные, неизменяемый указатель на изменяемые данные, изменяемый указатель на неизменяемые данные и неизменяемый указатель на неизменяемые данные. Каждая из четырех комбинаций обеспечивает различный уровень привилегий доступа к данным.

Самый высокий уровень доступа предоставляется не-константным указателям на не-константные данные. В этом случае данные могут изменяться через разыменование указателя, а сам указатель может изменяться и ссылаться на другие элементы данных. Объявление указателя — не константы на не-константные данные не содержит ключевого слова **const**. Такой указатель мог бы использоваться, например, при передаче строки как параметра функции, в которой используются арифметические операции над указателем при обработке (и, возможно, изменении) каждого отдельного символа в строке. Функция **convertToUppercase**, приведенная на рис. 7.10, объявляет параметром указатель — не константу на не-константные данные (**char * s**). Функция обрабатывает строку s посимвольно, используя арифметические операции над указате-

лем. Если символ находится в диапазоне от **a** до **z**, то он преобразуется в соответствующий символ верхнего регистра от **A** до **Z** с помощью вычислений, основанных на ASCII-кодах символов; в противном случае символ не изменяется и обрабатывается следующий символ в строке. Заметим, что все символы верхнего регистра имеют ASCII-код, равный ASCII-коду соответствующего символа нижнего регистра минус 32 (см. ASCII-таблицу в приложении Г). В главе 8 мы представим вам функцию **toupper** из стандартной библиотеки C, используемую для преобразования символов в символы верхнего регистра.

```
/* Преобразование символов нижнего регистра в символы верхнего */
/* регистра с использованием указателя-не константы на данные, */
/* которые могут модифицироваться */
#include <stdio.h>

void convertToUppercase(char *);

main()

{
   char string[] = "characters";

   printf("The string before conversion is: %s\n", string);
   convertToUppercase(string);
   printf("The string after conversion is:  %s\n", string);
   return 0;
}

void convertToUppercase(char *s)
{
   while (*s ! = '\0' ) {

      if (*s >= 'a' && *s <= 'z')
         *s -= 32;  /* преобразование к верхнему регистру */

      ++s;  /* увеличение s для перехода к следующему символу */
   }
}
```

```
The string before conversion is: characters
The string after conversion is:  CHARACTERS
```

Рис. 7.10. Преобразование строки к верхнему регистру с использованием не константного указателя на не константные данные

Указатель — не константа на данные-константы — это указатель, который может изменяться и ссылаться на любой объект данных соответствующего типа, но элемент данных, на которые он указывает, не может изменяться. Такой указатель мог бы использоваться для передачи массива в качестве аргумента функции, которая обрабатывает каждый элемент массива и при этом не изменяет данные. Например, приведенная на рис. 7.11 функция **printCharacters** объявляет параметр **s**, типа **const char ***. Объявление читается справа налево, и звучит как «**s** — указатель на символ-константу». В теле функции для вывода символов строки используется цикл **for**, который завершается при достижении конца строки. После вывода очередного символа производится увеличение указателя для перехода к следующему символу в строке.

```
/ * Посимвольный вывод строки, использующий */
/ * не-константный указатель на константные данные * /
#include <stdio.h>

void printCharacters(const char *);

main()
{
   char string[] = "print characters of a string";

   printf("The string is:\n");
   printCharacters(string);
   putchar('\n');
   return 0;
}

void printCharacters(const char *s)
{
   for ( ; *s != '\0'; s++)    /* цикл не инициализируется */
      putchar(*s);
}
```

```
The string is:
print characters of a string
```

Рис. 7.11. Посимвольный вывод строки с использованием указателя — не константы на константные данные

На рис. 7.12 приведены сообщения об ошибках, которые выдает компилятор Borland C ++ при попытке компилировать функцию, получающую указатель — не константу на данные-константы и пытающуюся при помощи указателя изменить эти данные.

Как мы знаем, массив является составным типом данных, объединяющим под одним именем логически связанные данные одного типа. В главе 10 мы обсудим другую форму составного типа данных, называемую *структурой* (или *записью* в других языках программирования). Структура объединяет связанные элементы данных различных типов под одним именем (например, информацию относительно каждого служащего компании). Когда в качестве параметра функции используется массив, то автоматически генерируется вызов по ссылке. Однако структуры всегда передаются по значению, т.е. передается копия всей структуры. Во время выполнения программы это приводит к дополнительным накладным расходам для создания копии каждого элемента данных в структуре и сохранении копии структуры в стеке компьютера. При передаче структуры в функцию мы можем использовать указатель на константные данные, чтобы сочетать эффективность вызова по ссылке с уровнем защиты данных, соответствующим вызову по значению. При передаче указателя на структуру функция будет получать только копию адреса структуры. При этом на машине с адресами в 4 байта копирование 4 байт адреса будет выполняться быстрее, чем копирование, возможно, сотен или тысяч байт элементов структуры.

Совет по повышению эффективности 7.1

При передаче больших объектов данных, например, структур, используйте указатель на константные данные. Этот подход позволит вам совместить производительность передачи параметра по ссылке и защиту данных как при передаче параметра по значению.

```
/*  Попытка изменения данных при помощи */
/* неконстантного указателя на  данные-константы */
#include <stdio.h>

void f(const int *);

main ( )
{
   int y;

   f(&y);      /* функция f пытается выполнить изменение данных */
   return 0;
}

void f(const int *x)
{
   *x = 100; /* нельз модифицировать объект данных типа const */
}
```

```
Compiling FIG7_12.C:
Error FIG7_12.C 17: Cannot modify a const object
Warning FIG7_12.C 18: Parameter 'x' is never used
```

Рис. 7.12. Попытка изменить данные, используя неконстантный указатель на константные данные

Такое применение указателей на константные данные — очередной пример компромисса время выполнения/требуемая память. Если памяти недостаточно и эффективность выполнения программы является важным фактором — используйте указатели. Если памяти достаточное количество, а бороться за эффективность не нужно, то, в соответствии с принципом минимума привилегий, данные должны передаваться по значению. Следует заметить, что некоторые системы недостаточно строго поддерживают модификатор **const**, так что передача параметра по значению остается самым надежным способом защиты данных от изменения.

Указатель-константа на не-константные данные — указатель, который всегда указывает на одно и то же место в памяти, а расположенные там данные могут изменяться. Такой указатель назначается по умолчанию для имени массива. Имя массива является указателем-константой на начало массива. Ко всем элементам массива можно обращаться и изменять их, используя имя массива и индекс. Константный указатель на не-константные данные может использоваться для передачи массива как параметра функции, которая обращается к его элементам, используя только индекс массива. Указатели, которые объявляются с модификатором **const**, должны быть инициализированы при объявлении (если такой указатель является параметром функции, он инициализируется переданным функции значением).

Программа на рис. 7.13 делает попытку изменить указатель-константу. Указатель **ptr** объявлен как **int * const**. Объявление должно читаться справа налево и в данном случае будет звучать как «**ptr** — указатель-константа на целое число». Указатель инициализирован адресом переменной **x** целого типа. Программа пытается присвоить указателю **ptr** адрес **y**, в результате чего выдается сообщение об ошибке.

```
/* Попытка изменения указателя-константы на */
/* не-константные данные  */
#include <stdio.h>

main()
{
   int x, y;
   int * const ptr = &x;

   ptr = &y;
   return 0;
}
```

```
Compiling FIG7_13.C:
Error FIG7_13.C 10: Cannot modify a const object
Warning FIG7_13.C 12:  'ptr' is assigned a value that is
   never used
Warning FIG7_13.C 12:  'y' is declared but never used
```

Рис. 7.13. Попытка изменения указателя-константы, ссылающегося на не-константные данные

Минимальные права доступа предоставляет указатель-константа на константные данные. Такой указатель всегда указывает на то же самое место в памяти, а расположенные по этому адресу данные не могут модифицироваться. Этому варианту соответствует передача массива функции, которая только просматривает массив при помощи индекса и не изменяет его элементы. В программе на рис. 7.14 объявляется переменная-указатель **ptr** типа **const int * const**. Объявление читается справа налево и звучит так: «**ptr** — указатель-константа на целую константу». На рисунке показаны сообщения об ошибках, полученных в результате попытки изменить данные, на которые указывает **ptr**, и изменить присвоенный указателю адрес.

```
/* Попытка модификации указателя-константы на */
/* данные-константы  */
#include <stdio.h>

main()
{
   int x = 5, y;
   const int *const ptr = &x;

   *ptr = 7;
   ptr = &y;
   return 0;
}
```

```
Compiling FIG7_14.C:
Error FIG7_14.C 10: Cannot modify a const object
Error FIG7_14.C 11: Cannot modify a const object
Warning FIG7_14.C 13:  'ptr' is assigned a value that is
   never used
Warning FIG7_14.C 13:  'y' is declared but never used
```

Рис. 7.14. Попытка изменить константный указатель на константные данные

7.6. Программа пузырьковой сортировки, использующая вызов по ссылке

Давайте изменим программу пузырьковой сортировки, приведенную на рис. 6.15, и введем две функции — **bubbleSort** и **swap**. Функция **bubbleSort** выполняет сортировку массива. Она вызывает функцию **swap** для перестановки элементов массива **array[j]** и **array[j + 1]** (см. рис. 7.15).

Как известно, функции в C скрывают свои данные друг от друга, поэтому **swap** не имеет доступа к элементам массива в функции **bubbleSort**. Но поскольку **bubbleSort** заинтересована в том, чтобы функция **swap** имела доступ к массиву, **bubbleSort** передает ей нужные элементы по ссылке, т.е. явно передает адреса элементов массива. При использовании имени массива в качестве аргумента передается по ссылке весь массив целиком, индивидуальные же элементы массива являются скалярами и передаются по значению. Поэтому **bubbleSort** при обращении к функции **swap** использует операцию взятия адреса (&) каждого из элементов:

```
swap (&array[j], &array[j + 1]);
```

чтобы передать их по ссылке. Функция **swap** помещает адрес **&array[j]** в переменную-указатель **element1Ptr**. И теперь, хотя **swap** и не имеет доступа к элементам массива **array[j]**, по причине сокрытия информации, **swap** может использовать разыменованный указатель ***element1Ptr** как синоним **array[j]**. Следовательно, когда **swap** обращается к ***element1Ptr**, она получает элемент **array[j]** из **bubbleSort**. Аналогично, когда **swap** ссылается на ***element2Ptr**, то получает **array[j + 1]** из **bubbleSort**.

И хотя **swap** не может выполнить следующие операторы,

```
temp = array[j];
array[j] = array[j + 1];
array[j + 1] = temp;
```

она получит тот же самый результат при помощи операторов

```
temp = *element1Ptr;
*element1Ptr = *element2Ptr;
*element2Ptr = temp;
```

в своем теле — см. рис. 7.15.

Отметим несколько особенностей функции **bubbleSort**. Ее заголовок объявляет массив **array** как **int *array**, а не как **int array[]**, поскольку эти способы записи параметра-массива взаимозаменяемы. В соответствии с принципом минимума привилегий, параметр **size** объявляется с модификатором **const**. И хотя параметр **size** содержит копию значения из функции **main**, изменение которой никак не может изменить значение переменной в **main**, функции **bubbleSort** не требуется изменять величину **size** для выполнения своей задачи. Размер массива остается величиной, постоянной во время выполнения **bubbleSort**. Поэтому параметр **size** и объявлен константным, чтобы гарантировать его неизменность. Если бы в процессе сортировки размер массива изменился, то, возможно, алгоритм выполнялся бы неправильно.

Прототип функции **swap** включен в тело функции, потому что **bubbleSort** — единственная функция, которая вызывает **swap**. Размещение прототипа в **bubbleSort** ограничивает правильные вызовы **swap** только теми, что были сделаны из **bubbleSort**. Другие функции, которые, не имея доступа к соответствующему функциональному прототипу, попытаются вызвать **swap**, будут использовать прототип по умолчанию. Обычно этот прототип не соответствует

заголовку функции (что приводит к ошибке компиляции), так как компилятор по умолчанию предполагает тип **int** для возвращаемого значения и параметров функции.

```
/* Эта программа заполняет массив, сортирует значения в порядке
   возрастания и выводит отсортированный массив */
#include <stdio.h>
#define SIZE 10

void bubbleSort(int *, int);

main()
{
   int i, a[SIZE] = {2, 6, 4, 8, 10, 12, 89, 68, 45, 37};

   printf("Data items in original order \n");

   for (i = 0; i <= SIZE - 1; i++)
      printf("%4d", a[i]);

   bubbleSort(a, SIZE);                /* сортировка массива */
   printf("\n Data items in ascending order\n");

   for (i = 0; i <= SIZE - 1; i++)
      printf("%4d", a[i]);

   printf("\n");
   return 0;
}

void bubbleSort(int *array, int size)
{
   int pass, j;
   void swap(int *, int *);
   for (pass = 1; pass <= size - 1; pass++)

      for (j = 0; j <= size - 2; j++)

         if (array[j] > array[j + 1])
            swap(&array[j], &array[j + 1]);
}

void swap(int *element1Ptr, int *element2Ptr)
{
   int temp;

   temp = *element1Ptr;
   *element1Ptr = *element2Ptr;
   *element2Ptr = temp;
}
```

```
Data items in original order
   2   6   4   8  10  12  89  68  45  37
Data items in ascending order
   2   4   6   8  10  12  37  45  68  89
```

Рис. 7.15. Пузырьковая сортировка и вызов по ссылке

Общее методическое замечание 7.4

Размещение прототипа функции в теле другой функции ограничивает корректные вызовы первой функции только теми, что будут сделаны из функции, где объявлен прототип. Это опять принцип минимума привилегий.

Отметим еще раз, что функция **bubbleSort** получает в качестве параметра размер массива. Для того, чтобы сортировать массив, функция должна знать его размер. Когда массив передается в функцию, функция получает адрес первого элемента. Адрес не несет в себе информации относительно числа элементов в массиве. Следовательно, программист должен каким-либо образом сообщить функции размер массива.

Передача функции размера массива в качестве параметра имеет два преимущества: во-первых — это хороший стиль программирования, во-вторых — такую функцию можно использовать многократно. Функция, которая получает размер массива как параметр, может быть использована любой другой программой, которая сортирует одномерные целочисленные массивы произвольного размера.

Общее методическое замечание 7.5

При передаче в функцию массива следует передавать и его размер. Это сделает функцию более универсальной. Универсальные функции могут использоваться большим количеством программ.

Мы могли бы поместить размер массива в глобальную переменную, которая была бы доступна для всей программы. Такой способ более эффективен, потому что не создается копия **size** для передачи в функцию. Но не все программы, которым требуется сортировать целочисленный массив, могут иметь глобальную переменную с нужным именем, и в таких программах функция не могла бы использоваться.

Общее методическое замечание 7.6

Глобальные переменные нарушают принцип минимальных привилегий и являются примером плохого стиля программирования.

Совет по повышению эффективности 7.2

Передача параметра требует времени и места в стеке для создания копии, передаваемой в функцию. Глобальным переменным не требуется этих дополнительных ресурсов, так как они доступны в любом месте программы.

Размер массива можно было бы «зашить» непосредственно в саму функцию. Но это ограничило бы применение функции только массивами специфического размера, т.е. чрезвычайно уменьшило бы возможность использования функции другими программами. Эта функция годилась бы только для программ, обрабатывающих одномерные целочисленные массивы специфического размера, запрограммированного в теле функции.

В C имеется специальная унарная операция **sizeof**, при помощи которой можно определить размер массива (или любого другого типа данных) в байтах во время компиляции программы. Если эту операцию применить к имени мас-

сива, как это показано на рис. 7.16, то **sizeof** возвратит общее количество байт, занимаемое массивом, в виде целого числа. Например, переменная типа **float** обычно занимает 4 байта памяти, а массив объявлен двадцатиэлементным. Следовательно, общее количество байт в массиве равно 80.

```
/* применение операции sizeof к массиву */
/* возвращается число байт в массиве */
#include <stdio.h>

main()
{
   float array[20];

   printf("The number of bytes in the array is %d\n",
          sizeof(array));

   return 0;
}
```

```
The number of bytes in the array is 80
```

Рис. 7.16. Операция **sizeof**, применяемая к имени массива, возвращает его размер в байтах

Число элементов в массиве также может быть определено во время компиляции. Например, рассмотрим следующее объявление массива:

```
double real[22];
```

Переменные типа **double** обычно занимают 8 байт памяти. Таким образом, массив **real** содержит 176 байт. Для того, чтобы определить число элементов в массиве, можно использовать следующее выражение:

```
sizeof(real) / sizeof(double)
```

В выражении определяется число байт в массиве **real** и полученное значение делится затем на число байт, используемых для представления типа **double**.

Программа на рис. 7.17 вычисляет число байт, используемых для представления стандартных типов данных совместимого с PC компьютера.

Совет по переносимости программ 7.2

Размер в байтах одного и того же типа данных может быть разным на различных системах. При написании программ, которые будут работать на различных платформах и при этом небезразличны к представлению данных, используйте операцию **sizeof** для определения размера типа данных.

Операция **sizeof** может применяться к любой переменной, типу данных или константе. При определении размера переменной или константы (но не массива) возвращается число байт, отводимых под тип переменной или константы. Обратите внимание, что скобки с **sizeof** обязательны, если в качестве операнда используется имя типа. Отсутствие скобок в этом случае приводит к сообщению об ошибке. Скобки не требуются, если операнд является именем переменной.

```
/* Использование операции sizeof * /
#include <stdio.h>

main()
{
  printf("        sizeof(char) = %d\n"
         "       sizeof(short) = %d\n"
         "         sizeof(int) = %d\n"
         "        sizeof(long) = %d\n"
         "       sizeof(float) = %d\n"
         "      sizeof(double) = %d\n"
         "sizeof(long double) = %d\n",
         sizeof(char), sizeof(short), sizeof(int),
         sizeof(long), sizeof(float), sizeof(double),
         sizeof(long double));
  return 0;
}
```

```
        sizeof(char) = 1
       sizeof(short) = 2
         sizeof(int) = 2
        sizeof(long) = 4
       sizeof(float) = 4
      sizeof(double) = 8
sizeof(long double) = 10
```

Рис. 7.17. Использование операции **sizeof** для определения размера стандартных типов данных

7.7. Выражения и арифметические операции с указателями

Указатели являются допустимыми операндами в арифметических выражениях, выражениях присваивания и сравнения. Однако не все операции, обычно используемые в этих выражениях, дают верный результат для переменных-указателей. Этот раздел описывает операции, которые могут применяться к указателям, и как эти операции должны использоваться.

К указателям может применяться ограниченный набор арифметических операций. Указатель может быть увеличен (++) или уменьшен (--), к указателю может быть прибавлено целое число (+ или +=), из указателя можно вычесть целое число (- или -=) и можно вычислить разность двух указателей.

В качестве примера определим массив **int v[10]**, первый элемент которого будет иметь адрес в памяти, равный **3000**. Инициализируем указатель **vPtr** значением адреса **v[0]**, т. е. значение **vPtr** равно **3000**. На рис. 7.18 этот пример изображен графически для машины с 4-байтными целыми. Заметим, что для того, чтобы **vPtr** указывал на массив **v**, его можно инициализировать любым из следующих операторов:

```
vPtr = v;
vPtr = &v[0];
```

В обычной арифметике результатом сложения **3000 + 2** будет значение **3002**. В арифметике указателей такой результат обычно будет неверным. При прибавлении или вычитании из указателя целого числа значение его увеличи-

вается или уменьшается не на это число, а на произведение числа на размер объекта, на который указатель ссылается.

Совет по переносимости программ 7.3

Большинство компьютеров сегодня поддерживают 2-байтовые и 4-байтовые целые. Некоторые из последних машин имеют еще и 8-байтовые целые. Поскольку результат арифметики указателей зависит от размера объектов, на которые ссылается указатель, то результат арифметических выражений с указателями является машинно-зависимым.

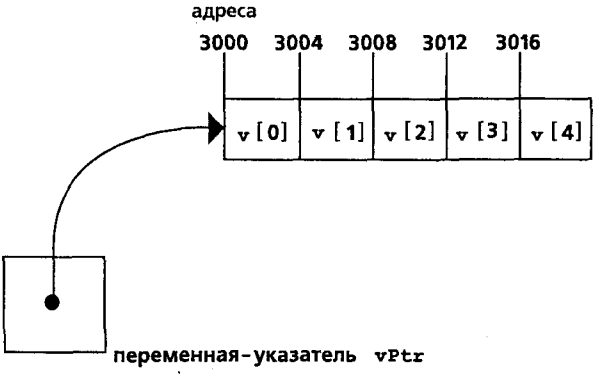


Рис. 7.18. Массив **v** и указатель **vPtr**, ссылающийся на v

Размер объекта в байтах зависит от типа объекта. Например, оператор

```
vPtr += 2;
```

даст результат **3008 (3000 + 2 * 4)**, если для целого числа отводится в памяти 4 байта. Теперь **vPtr** будет ссылаться на элемент **v[2]** (см рис. 7.19). Если бы целое число занимало 2 байта, то предыдущий расчет дал бы адрес в памяти **3004 (3000 + 2 * 2)**. Если бы массив состоял из данных другого типа, предыдущий оператор увеличил бы указатель на удвоенный размер объекта данного типа. При выполнении арифметических операций с указателем на символьный массив результаты будут совпадать с арифметическими, потому что под каждый символ отводится в памяти один байт.

Если бы **vPtr** был увеличен до значения **3016**, которое соответствует адресу элемента массива **v[4]**, то оператор

```
vPtr -= 4;
```

вернул бы **vPtr** к значению **3000**, соответствующему началу массива. При увеличении или уменьшении указателя на единицу можно использовать операции инкремента (++) и декремента (--). Каждый из следующих операторов

```
++vPtr;
vPtr++;
```

увеличивает значение указателя, который будет ссылаться на следующий элемент массива. Любой из следующих операторов

```
--vPtr;
vPtr--;
```

уменьшает значение указателя, который получает при этом доступ к предыдущему элементу массива.

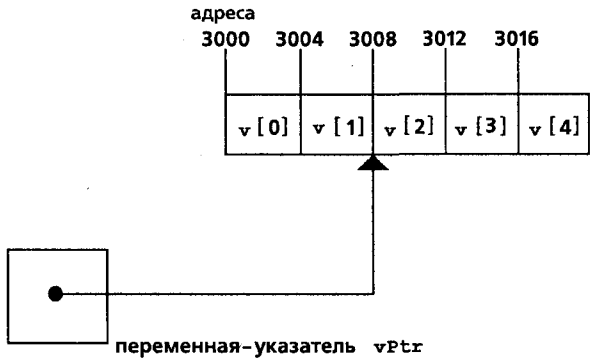


Рис. 7.19. Указатель **vPtr** после арифметического преобразования

Переменные-указатели могут вычитаться друг из друга. Например, если значение **vPtr** равно **3000**, а **v2Ptr** содержит адрес **3008**, то в результате выполнения оператора

```
x = v2Ptr - vPtr;
```

переменной x будет присвоено число элементов массива, расположенных начиная с адреса **vPtr** и до **v2Ptr**; в нашем случае это будет значение **2**. Обычно арифметические операции с указателями используются при работе с массивами. Элементы массива хранятся последовательно, друг за другом, а две переменные одного и того же типа не обязательно находятся в памяти рядом.

Распространенная ошибка программирования 7.5

Использование арифметических операций с указателем, ссылающимся не на элементы массива.

Распространенная ошибка программирования 7.6

Вычитание или сравнение двух указателей, ссылающихся не на один и тот же массив.

Распространенная ошибка программирования 7.7

Выход за начало или конец массива при арифметических операциях с указателем.

Указатель может быть присвоен другому указателю, если оба указателя имеют один и тот же тип. В противном случае нужно использовать операцию приведения типа указателя в правой части оператора присваивания к типу указателя в левой части. Исключением из этого правила является указатель на **void** (т.е. типа **void ***), который является обобщенным указателем и может представлять любой тип указателя. Указатель любого типа может быть присвоен указателю на **void**, и **void**-указатель может быть присвоен указателю любого типа. В обоих случаях применение операции приведения типа не требуется.

Указатель на **void** не может быть разыменован. Например, при разыменовании указателя на целое компилятор знает, что тот ссылается на четыре байта памяти (на машине с целыми числами размером в 4 байта), но **void**-указатель содержит адрес памяти для неизвестного типа данных, размер которого

не известен компилятору. Компилятор должен знать тип данных и, тем самым, размер элемента данных в байтах, чтобы правильно разыменовать указатель. В случае указателя на **void** размер элемента в байтах не может быть определен компилятором.

Распространенная ошибка программирования 7.8

Присвоение значения указателя одного типа указателю другого типа, когда ни один из них не является указателем типа **void** *, приводит к синтаксической ошибке.

Распространенная ошибка программирования 7.9

Разыменование **void**-указателя.

Указатели могут сравниваться друг с другом при помощи операций сравнения и отношения, но сравнение указателей обычно не имеет смысла, если они не ссылаются на элементы одного и того же массива. При сравнении указателей сравниваются адреса, являющиеся значениями указателей. Сравнение двух указателей, ссылающихся на элементы одного и того же массива, могло бы показать, например, что один указатель ссылается на элемент с большим значением индекса, чем другой указатель. Другая часто используемая операция сравнения указателя — это проверка, не равно ли его значение **NULL**.

7.8. Связь между указателями и массивами

Массивы и указатели в C тесно связаны друг с другом и практически являются взаимозаменяемыми. Имя массива можно рассматривать как указатель-константу. А над указателями можно выполнять различные операции, в том числе использовать с указателем индексные выражения.

Совет по повышению эффективности 7.3

Ссылка на элемент массива при помощи индекса преобразуется во время компиляции в выражение с указателем, поэтому, если вы сами будете обращаться к элементам массива при помощи указателей, то сократите время компиляции вашей программы.

Хороший стиль программирования 7.5

Ссылайтесь на элементы массива при помощи индексов. Пусть ваша программа будет компилироваться немного дольше, но зато текст программы будет более понятен.

В качестве примера объявим целочисленный массив **b[5]** и переменную-указатель **bPtr** на целое. Так как имя массива (если с ним не указан индекс) является указателем на первый элемент массива, мы можем присвоить указателю **bPtr** адрес первого элемента массива **b** при помощи оператора присваивания

```
bPtr = b;
```

Этот оператор эквивалентен следующему оператору, в котором используется операция взятия адреса первого элемента массива:

```
bPtr = &b[0];
```

Альтернативный способ ссылки на элемент массива **b[3]**, использующий выражение с указателем, представлен в следующем операторе:

```
*(bPtr + 3)
```

Константа **3** в приведенном выражении называется *смещением*. Когда указатель ссылается на начало массива, величина смещения указывает, на какой элемент массива производится ссылка; значение смещения равно значению индекса массива. Приведенный способ записи носит название нотации *указатель/смещение*. В этом выражении использованы круглые скобки, потому что операция * имеет больший приоритет, чем операция +. Без круглых скобок в вышеупомянутом выражении значение 3 было бы прибавлено к значению выражения ***bPtr** (т.е. число **3** будет прибавлено к элементу **b[0]**, так как **bPtr** указывает на начало массива). Поскольку на значение элемента массива можно сослаться при помощи выражения с указателем, адрес элемента **&b[3]** представляется выражением

```
bPtr + 3
```

Имя массива может рассматриваться как указатель, так что его можно использовать в выражениях арифметики указателей. Например, выражение

```
*(b + 3)
```

будет ссылаться на элемент массива **b[3]**. Вообще говоря, все выражения с индексами могут быть преобразованы в выражения с указателем и смещением. В этом случае в качестве указателя можно использовать имя массива. Обратим внимание на то, что в предыдущем значение указателя выражении не изменяется **b**, он по-прежнему указывает на первый элемент в массиве.

Указатели, в свою очередь, могут быть использованы вместо имен массивов в индексных выражениях. Например, выражение

```
bPtr[1]
```

представляет собой ссылку на элемент массива **b[1]**. Такой способ записи можно назвать методом указатель/индекс.

Не забудьте, что имя массива — это указатель-константа и он всегда указывает на начало массива. Поэтому выражение

```
b += 3
```

является недопустимым, так как в нем делается попытка изменить значение начального адреса массива.

Распространенная ошибка программирования 7.10

Попытка изменения имени массива, являющегося указателем-константой, приводит к синтаксической ошибке.

В программе на рис. 7.20 используются четыре метода ссылки на элементы массива, которые мы обсудили — имя массива и индекс, указатель/смещение с именем массива, указатель и индекс и указатель/смещение, — для вывода четырех элементов целочисленного массива **b**.

В качестве еще одного примера взаимозаменяемости массивов и указателей рассмотрим две функции копирования строк — **copy1** и **copy2** — в программе на рис. 7.21. Обе функции копируют строку символов (возможно, символьный массив) в массив символов. Прототипы функций **copy1** и **copy2** абсолютно идентичны. И действительно, они выполняют одну и ту же задачу, но делают это по-разному.

```
/* Использование индексов и указателей при работе с массивами */
#include <stdio.h>

main ( )
{
   int i, offset, b[] = {10, 20, 30, 40};
   int *bPtr = b;    /* указателю bPtr присваивается начальный
                        адрес массива b */

   printf("Array b is printed with:\n"
          "Array subscript notation\n");

   for (i = 0; i <= 3; i++)
      printf ("b[%d] = %d\n", i, b[i] );

   printf ( "\nPointer/offset notation where \n"
            "the pointer is the array name\n");

   for (offset = 0; offset <= 3; offset++)
      printf("*(b + %d) = %d\n", offset, *(b + offset));

   printf ("\nPointer subscript notation\n");

   for (i = 0; i <= 3; i++)
      printf( bPtr[%d] = %d\n , i, bPtr[i]);

   printf("\nPointer/offset notation\n");

   for (offset = 0; offset <= 3; offset++)
      printf("*(bPtr + %d) = %d\n", offset, *(bPtr + offset));

   return 0;
}
```

```
Array b is printed with:
Array subscript notation
b[0] = 10
b[1] = 20
b[2] = 30
b[3] = 40

Pointer/offset notation where
the pointer is the array name
*(b + 0) = 10
*(b + 1) = 20
*(b + 2) = 30
*(b + 3) = 40

Pointer subscript notation
bPtr[0] = 10
bPtr[1] = 20
bPtr[2] = 30
bPtr[3] = 40
```

Рис. 7.20. Использование четырех методов ссылки на элементы массива (часть 1 из 2)

```
Pointer/offset notation
*(bPtr + 0) = 10
*(bPtr + 1) = 20
*(bPtr + 2) = 30
*(bPtr + 3) = 40
```

Рис. 7.20. Использование четырех методов ссылки на элементы массива (часть 2 из 2)

```
/* Копирование строки с использованием индексов с именем массива
   и арифметики указателей */
#include <stdio.h>

void copy1(char *, const char *);
void copy2(char *, const char *);

main()

{
   char string1[10], *string2 = "Hello",
        string3[10], string4[] =  "Good bye";

   copy1(string1, string2);
   printf("string1 = %s\n", string1);

   copy2(string3, string4);
   printf("string3 = %s\n", string3);
   return 0;
}

/* копирование s2 в s1 с помощью индексной нотации */
void copy1(char *s1, const char *s2)
{
   int i;

   for (i = 0; s1[i] = s2[i]; i++)
      ;              /* тело цикла пусто */
}

/* копирование s2 в s1 с использованием арифметики указателей */
void copy2(char *s1, const char *s2)
{
   for (; *s1 = *s2; s1++, s2++)
      ;              /* тело цикла пусто */
}
```

```
string1 = "Hello",
string3 =  "Good bye";
```

Рис. 7.21. Копирование строки с помощью индексов и указателей

Функция **copy1** применяет для копирования строки **s2** в массив **s1** индексную нотацию. Функция объявляет целую переменную-счетчик **i**, используемую как индекс массива. В заголовке цикла **for** выполняется вся работа по созданию копии, при этом тело цикла пусто. В заголовке цикла **i** инициализируется нулем и увеличивается на единицу на каждом шаге цикла. Условие цик-

ла **for**, **s1[i] = s2[i]**, выполняет посимвольное копирование строки **s2** в **s1**. Когда в **s2** встречается нуль-символ, он присваивается **s1**, и цикл завершается, потому что целочисленное значение нуль-символа равно нулю (а значение выражения — **false**). Напомним, что значением оператора присваивания является значение, получаемое левым операндом.

Функция **copy2** при копировании строки **s2** в символьный массив **s1** использует указатели и арифметику указателей. Здесь также все операции копирования производятся в заголовке цикла **for**. В цикле не используется управляющая переменная. Как и в функции **copy1**, копирование символов строки выполняется в выражении условия цикла (***s1 = *s2**). Указатель **s2** разыменовывается и полученный символ присваивается разыменованному указателю **s1**. После выполнения присваивания в выражении условия цикла указатели увеличиваются для перехода соответственно к следующему элементу массива **s1** и следующему символу строки **s2**. Когда в строке **s2** встречается нуль-символ, он присваивается разыменованному указателю **s1** и цикл завершается.

Обратите внимание на то, что первый параметр функций **copy1** и **copy2** должен быть массивом достаточно большого размера, чтобы вместить строку, передаваемую вторым параметром. В противном случае произойдет ошибка при записи в область памяти, которая не является частью массива. Заметьте также, что второй параметр каждой функции объявлен как **const char *** (строка-константа). В обеих функциях второй аргумент копируется в первый аргумент, при этом символы из второго аргумента читаются по одному и не изменяются. Поэтому, в соответствии с принципом минимума привилегий, второй параметр объявляется указателем на значение-константу.

Ни одной из функций не нужно изменять второй аргумент, поэтому ни одной из них и не предоставляется такое право.

7.9. Массивы указателей

Массивы могут состоять из указателей. Обычный случай такого массива — это массив строк, который так и называется *массив строк*. Элементом такого массива является строка, а строки в C являются, по существу, указателями на первый символ строки. Значит, элементами строкового массива являются указатели на начала строк. В качестве примера рассмотрим массив **suit**, который может пригодится для описания игральных карт.

```
char *suit[4] = {"Hearts", "Diamonds", "Clubs", "Spades"};
```

Выражение **suit[4]** в объявлении означает массив из четырех элементов. При помощи **char *** этот массив объявляется состоящим из указателей на тип char. В массив помещаются четыре значения **«Hearts»**, **«Diamonds»**, **«Clubs»** и **«Spades»** (**«Червы»**, **«Бубны»**, **«Трефы»** и **«Пики»**). Каждое из этих значений хранится в памяти как строка символов с конечным нулем, длиной на один символ больше, чем количество символов, заключенных в кавычки. Строки эти занимают в памяти соответственно 7, 9, 6 и 7 байт. И хотя кажется, что в массив помещаются сами строки, элементами массива являются указатели (см. рис 7.22).

Каждый указатель ссылается на первый символ соответствующей строки. Таким образом, хотя массив **suit** имеет фиксированный размер, он может «хранить» символьные строки произвольной длины. Это еще один пример мощных возможностей структурирования данных в C.

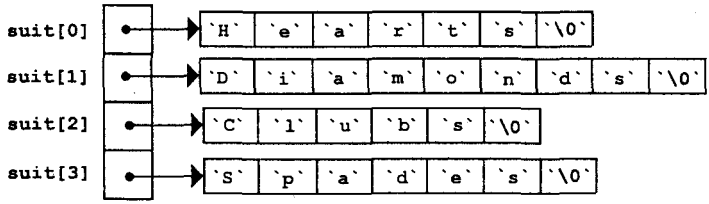


Рис. 7.22. Графическое представление массива **suit**

Рассматриваемый нами массив карточных мастей можно было бы сделать двумерным: имя каждой масти занимало бы одну строку, а в каждом столбце помещалось бы по одному символу имени масти. При таком подходе все строки массива должны быть одинаковой длины, равной размеру самой длинной строки символов. Это привело бы к неоправданному расходу памяти в случае, когда большинство сохраняемых строк короче, чем самая длинная строка. В следующем разделе мы будем использовать массивы строк для представления колоды карт.

7.10. Пример: программа тасовки и сдачи колоды карт

В этом разделе, используя генератор случайных чисел, мы составим программу, моделирующую тасовку и сдачу карт. Эту программу можно будет потом использовать при написании игровых карточных программ. Мы намеренно использовали неоптимальные алгоритмы тасования и сдачи, чтобы таким образом познакомить вас с некоторыми тонкими проблемами эффективности. В упражнениях и в главе 10 мы разработаем более эффективные алгоритмы.

Действуя методом нисходящего последовательного уточнения, мы разработаем программу, которая перетасует колоду из 52 игральных карт и затем все их раздаст. Нисходящий метод особенно полезен при решении больших и сложных проблем, а не таких простых, как те, с которыми мы имели дело в предыдущих главах.

Для представления колоды мы используем двумерный массив **deck** размером 4 на 13 (см. рис. 7.23). Строки массива соответствуют мастям; строка 0 — червы, строка 1 — бубны, строка 2 — трефы, и строка 3 представляет пики. По столбцам записаны номиналы карт, столбцы с индексом от 0 до 9 соответствуют картам соответственно от туза до десятки, а столбцы с 10-го по 12-й соответствуют вальту, даме и королю. Мы заполним массив **suit** строками — именами четырех мастей, а массив **face** — строками, соответствующими тринадцати карточным номиналам.

Эта воображаемая колода карт может быть перетасована следующим образом. Сначала массив **deck** заполняется нулями. Затем случайным образом выбирается номер строки **row** (число от 0 до 3) и номер столбца **column** (число в диапазоне 0–12). В выбранный таким образом элемент массива **deck[row][column]** помещается число 1, которое означает, что эта карта из «перетасованной» колоды будет сдана первой. Этот процесс продолжается далее для чисел 2, 3, ..., 52, вставляемых случайным образом в массив **deck** и обозначающих карты, которые должны быть сданы второй, третьей и т.д. до пятьдесят второй. Как только массив **deck** начинает заполняться числами, возникает веро-

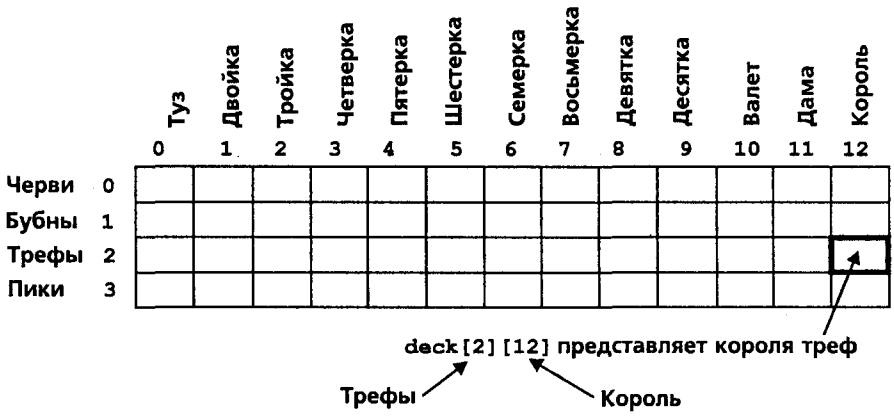


Рис. 7.23. Представление колоды карт в виде двумерного массива

ятность, что некоторые карты будут выбраны дважды, т.е. при выборе некоторого элемента **deck[row][column]** его значение будет отлично от нуля. В этом случае случайный выбор **row** и **column** повторяется до тех пор, пока не будет найдена карта, которая еще не выбиралась. В конечном счете числа от 1 до 52 займут все 52 элемента массива **deck**. Тем самым колода будет полностью перетасована.

Этот алгоритм перетасовывания может выполняться неопределенно долго, если карты, которые уже были сданы, продолжают выпадать в случайном выборе. Это явление известно и носит название *бесконечной отсрочки*. В упражнениях мы обсудим улучшенный алгоритм тасования колоды, в котором устранена возможность бесконечной отсрочки.

Совет по повышению эффективности 7.4

Часто «естественный» алгоритм может столкнуться с неожиданными препятствиями при своем выполнении, как, например, бесконечная отсрочка. Ищите алгоритмы, которые не содержат в себе таких «бесконечностей».

Чтобы начать сдачу колоды, мы должны найти элемент массива **deck[row][column]**, значение которого равно **1**. Эту задачу выполняют вложенные циклы **for**, в которых **row** изменяется от 0 до 3, а **column** — от 0 до 12. Как узнать, какая карта соответствует данному элементу массива? Так как массив мастей **suit** был уже нами определен, для того, чтобы получить название масти, нужно вывести строку **suit[row]**. Аналогично, чтобы получить название номинала карты, нам нужно вывести строку **face[column]**. Кроме того, мы выводим строку **" of "**. Информация относительно всех сданных карт выводится в форме **"King of Clubs"**, **"Ace of Diamonds"** и т.д.

Давайте приступим к процессу нисходящего проектирования. Сначала запишем нашу задачу в общем виде

Тасование и раздача 52 карт

На первом шаге детализации алгоритма его можно записать так:

Инициализация массива suit

Инициализация массива face
Инициализация массива deck
Тасование колоды
Раздача 52 карт

Пункт «Тасование колоды» может быть детализирован следующим образом:

Для каждой из 52 карт
    Записать номер карты в выбранный случайным образом свободный элемент массива deck

Пункт «Раздача 52 карт» может быть детализирован следующим образом:

Для каждой из 52 карт
    Найти номер карты в массиве deck и вывести название номинала и масти карты

С учетом сделанных расширений на втором шаге детализации алгоритм будет выглядеть так:

Инициализация массива suit
Инициализация массива face
Инициализация массива deck

Для каждой из 52 карт
    Записать номер карты в выбранный случайным образом свободный элемент массива deck

Для каждой из 52 карт
    Найти номер карты в массиве deck и вывести название номинала и масти карты

Пункт «Записать номер карты в выбранный случайным образом свободный элемент массива deck» может быть расширен следующим образом:

Выбрать случайным образом элемент массива deck
Пока выбранный элемент уже выбирался ранее
    Выбрать случайным образом элемент массива deck
Записать номер карты в выбранный элемент массива deck

Пункт «Найти номер карты в массиве deck и вывести название номинала и масти карты» можно детализировать как:

Для каждого элемента массива
    Если элемент содержит искомый номер карты
        Вывести название номинала карты и ее масть

В результате на третьем шаге детализации алгоритм примет вид:

Инициализация массива suit
Инициализация массива face
Инициализация массива deck

Для каждой из 52 карт
    Выбрать случайным образом элемент массива deck
    Пока выбранный элемент уже выбирался ранее
        Выбрать случайным образом элемент массива deck
    Записать номер карты в выбранный элемент массива deck

Для каждой из 52 карт
Для каждого элемента массива
Если элемент содержит искомый номер карты
Вывести название номинала карты и ее масть

На этом процесс детализации алгоритма можно завершить. Обратите внимание на то, что эту программу можно сделать более эффективной, если алгоритмы тасования и раздачи объединить и сразу сдавать помещенную в колоду карту. Мы выбрали раздельный вариант этих алгоритмов, поскольку он ближе к реальности: карты сначала тасуются, а потом сдаются.

Программа тасования и сдачи колоды карт приведена на рис. 7.24, а пример ее исполнения — на рис. 7.25. Обратите внимание на использование спецификатора формата **%s** функции **printf** для вывода строки символов. Соответствующий этому спецификатору аргумент функции **printf** должен быть указателем на тип **char** (или символьным массивом). В функции **deal** спецификация **"%5s of %-8s"** выводит символьную строку, выровненную по правому краю, в поле из пяти символов, за которой следует строка **" of "**, и строку символов, выровненную по левому краю, в поле из восьми символов. Знак **«минус»** в спецификации **%-8s** как раз и означает выравнивание по левому краю в поле шириной 8 символов.

```
/* Программа, сдающая колоду карт */

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

void shuffle(int [][13]);
void deal(const int [][13], const char *[], const char *[]);

main()
{
   char *suit [4] = {"Hearts", "Diamonds", "Clubs", "Spades"};
   char *face[13] = {"Ace", "Deuce", "Three", "Four",
                     "Five", "Six", "Seven", "Eight",
                     "Nine", "Ten", "Jack", "Queen", "King"};
   int deck[4][13] = {0};

   srand(time(NULL));

   shuffle(deck);
   deal(deck, face, suit);

   return 0;
}

void shuffle(int wDeck[][13])
{
   int card, row, column;

   for (card = 1; card <= 52; card++) {
      row = rand() % 4;
      column = rand() % 13;
```

Рис. 7.24. Программа раздачи колоды карт (часть 1 из 2)

```
      while (wDeck[row][column] != 0) {
         row = rand() % 4;
         column = rand() % 13;
      }

      wDeck[row][column] = card;
   }
}

void deal(const int wDeck[ ][13], const char *wFace[ ],
const char *wSuit[ ])
{
   int card, row, column;

   for (card = 1; card <= 52; card++)

      for (row = 0; row <= 3; row++)

         for (column = 0; column <= 12; column++)

            if (wDeck[row][column] == card)
               printf("%5s of %-8s%c",
                      wFace[column], wSuit[row],
                      card % 2 == 0 ? '\n' : '\t');
}
```

Рис. 7.24. Программа раздачи колоды карт (часть 2 из 2)

```
  Six of Clubs       Seven of Diamonds
  Ace of Spades        Ace of Diamonds
  Ace of Hearts      Queen of Diamonds
Queen of Clubs       Seven of Hearts
  Ten of Hearts      Deuce of Clubs
  Ten of Spades      Three of Spades
  Ten of Diamonds     Four of Spades
 Four of Diamonds      Ten of Clubs
  Six of Diamonds      Six of Spades
Eight of Hearts      Three of Diamonds
 Nine of Hearts      Three of Hearts
Deuce of Spades        Six of Hearts
 Five of Clubs       Eight of Clubs
Deuce of Diamonds    Eight of Spades
 Five of Spades       King of Clubs
 King of Diamonds     Jack of Spades
Deuce of Hearts      Queen of Hearts
  Ace of Clubs        King of Spades
Three of Clubs       King of Hearts
 Nine of Clubs        Nine of Spades
 Four of Hearts      Queen of Spades
Eight of Diamonds     Nine of Diamonds
 Jack of Diamonds    Seven of Clubs
 Five of Hearts       Five of Diamonds
 Four of Clubs        Jack of Hearts
 Jack of Clubs       Seven of Spades
```

Рис. 7.25. Пример выполнения программы сдачи колоды карт

У нашего алгоритма сдачи карт есть один недостаток. Когда найдена нужная карта, иногда с первой попытки, два внутренних цикла **for** продолжают поиск в оставшейся части массива **deck**. В упражнениях и в главе 10 мы исправим этот дефект.

7.11. Указатели на функции

Указатель на функцию — это переменная, содержащая адрес в памяти, по которому расположена функция. Из главы 6 мы знаем, что имя массива является адресом первого элемента массива. Аналогичным образом имя функции — это адрес начала программного кода функции. Указатели на функции могут быть переданы функциям в качестве аргументов, могут возвращаться функциями, сохраняться в массивах и присваиваться другим указателям на функции.

Чтобы проиллюстрировать использование указателей на функции, мы возьмем программу пузырьковой сортировки, приведенную на рис. 7.15, и создадим новый ее вариант, код которого приведен на рис. 7.26. Наша новая программа состоит из функций **main**, **bubble**, **swap**, **ascending** и **descending**. Функция **bubble** получает указатель на функцию — это может быть функция **ascending** или функция **descending** — в дополнение к двум другим параметрам: целочисленному массиву и размеру массива. Во время исполнения программа запрашивает у пользователя способ сортировки — в порядке возрастания или убывания. Если пользователь вводит число 1, функции **bubble** передается указатель на функцию **ascending** и производится сортировка переданного массива по возрастанию. Если пользователь вводит число 2, то в функцию **bubble** передается указатель на функцию **descending** и производится сортировка массива по убыванию. Пример двух сеансов работы с программой показан на рис. 7.27.

В заголовке функции **bubble** имеется следующий параметр:

```
int (* compare)(int, int)
```

Это значит, что функция **bubble** получает аргумент, являющийся указателем на функцию, которая имеет два целочисленных параметра и возвращает результат целого типа. Вокруг ***compare** необходимы круглые скобки, потому что операция * имеет более низкий приоритет, чем скобки, в которые заключены параметры функции. Если мы уберем круглые скобки, то объявление будет иметь вид

```
int * compare (int, int)
```

и будет обозначать функцию, которая получает два целых числа как параметры и возвращает указатель на целое число.

Прототип функции **bubble** может быть объявлен и в таком виде:

```
int (*) (int, int)
```

Обратите внимание на то, что обязательны только объявления типа данных. Программист может включать в прототип имена, чтобы текст программы был более ясным. Компилятор будет их игнорировать.

Функция, переданная функции **bubble**, вызывается в операторе **if** следующим образом:

```
if ((*compare) (work [count], work [count + 1]))
```

Подобно тому, как разыменовывают указатель на переменную, чтобы получить ее значение, указатель на функцию разыменовывают, чтобы произвести вызов этой функции.

```
/* Многоцелевая программа сортировки, использующая указатели
   на функцию */
#include <stdio.h>
#define SIZE 10

void bubble(int *, const int, int (*)(int, int));
int ascending(const int, const int);
int descending(const int, const int);

main()
{
   int a[SIZE] = {2, 6, 4, 8, 10, 12, 89, 68, 45, 37};
   int counter, order;

   printf("Enter 1 to sort in ascending order,\n");
   printf("Enter 2 to sort in descending order: ");
   scanf("%d", &order);

   printf("\nData items in original order\n");

   for (counter = 0; counter <= SIZE - 1; counter++)
      printf("%4d", a[counter]);

   if (order == 1) {
      bubble(a, SIZE, ascending);
      printf("\nData items in ascending order\n");
   }
   else {
      bubble(a, SIZE, descending);
      printf("\nData items in descending order\n");
   }

   for (counter = 0; counter <= SIZE - 1; counter++)
      printf("%4d", a[counter]);

   printf ("\n" );

   return 0;
}

void bubble(int *work, const int size, int (*compare)(int, int))
{
   int pass, count;
   void swap(int *, int *);

      for (pass = 1; pass <= size - 1; pass++)

         for (count = 0; count <= size - 2; count++)

            if ((*compare)(work[count], work[count + 1]))
               swap(&work[count], &work[count + 1]);

}
```

Рис. 7.26. Многоцелевая программа сортировки, использующая указатели на функцию (часть 1 из 2)

```
void swap(int *element1Ptr, int *element2Ptr)
{
   int temp;

   temp = *element1Ptr;
   *element1Ptr = *element2Ptr;
   *element2Ptr = temp;
}

int ascending(const int a, const int b)
{
   return b < a;
}

int descending(const int a, const int b)
{
   return b > a;
}
```

Рис. 7.26. Многоцелевая программа сортировки, использующая указатели на функцию (часть 2 из 2)

```
Enter 1 to sort in ascending order,
Enter 2 to sort in descending order: 1

Data items in original order
2   6   4   8  10  12  89  68  45  37
Data items in ascending order
2   4   6   8  10  12  37  45  68  89

Enter 1 to sort in ascending order,
Enter 2 to sort in descending order: 2

Data items in original order
2   6   4   8  10  12  89  68  45  37
Data items in descending order
89 68 45  37 12  10   8   6   4   2
```

Рис. 7.27. Результаты работы программы пузырьковой сортировки, приведенной на рис. 7.26

Функцию можно вызвать и без разыменования указателя, как это сделано в следующей строке:

```
if (compare(work[count], work[count + 1]))
```

Здесь указатель используется непосредственно, как имя функции. Мы предпочитаем первый метод вызова — с разыменованием указателя, потому что в этом варианте явно видно, что **compare** является указателем на функцию. При втором методе вызова функции через указатель может создаться впечатление, что **compare** — это имя функции. В результате пользователь программы может запутаться, если попытается найти определение функции **compare** и обнаружит, что она нигде в файле не определена.

Указатели на функции часто используются в системах, управляемых посредством меню. Пользователь выбирает команду меню (например, одну из пяти). Каждая команда обслуживается своей функцией. Указатели на каждую функцию находятся в массиве указателей. Выбор пользователя служит индексом, по которому из массива выбирается указатель на нужную функцию.

Программа на рис. 7.28 показывает типовой пример объявления и использования массива указателей на функцию. Определяются три функции — **function1**, **function2** и **function3**, — каждая из которых имеет целочисленный параметр и не возвращает значения. Указатели на эти три функции находятся в массиве **f**, который объявляется так:

```
void (*f[3] ) (int) = (function1, function2, function3);
```

Объявление читается, начиная с самой левой скобки и звучит как: «**f** — массив из трех указателей на функцию, которая имеет параметр целого типа и не возвращает значения». Массив инициализируется именами трех функций. Когда пользователь вводит значение от 0 до 2, это значение используется в качестве индекса массива указателей на функцию. Вызов функции выполняется следующим образом:

```
(*f[choice])(choice);
```

```
/*Пример использования массива указателей на функцию */
#include <stdio.h>

void function1(int);
void function2(int);
void function3(int);

main()
{
   void (*f[3])(int) = (function1, function2, function3);
   int choice;

   printf("Enter a number between 0 and 2, 3 to end: ");
   scanf("%d", &choice);

   while (choice >= 0 && choice < 3) {
      (*f[choice])(choice);
      printf("Enter a number between 0 and 2, 3 to end: ");
      scanf("%d", &choice);
   }

   printf("You entered 3 to end\n");
   return 0;
}

void function1(int a)
{
   printf("You entered %d so function1 was called\n\n", a);
}

void function2(int b)
{
   printf("You entered %d so function2 was called\n\n", b);
}

void function3(int c)
{
   printf("You entered %d so function3 was called\n\n", c);
}
```

Рис. 7.28. Пример массива указателей на функцию (часть 1 из 2)

```
Enter a number between 0 and 2, 3 to end: 0
You entered 0 so function1 was called

Enter a number between 0 and 2, 3 to end: 1
You entered 1 so function2 was called

Enter a number between 0 and 2, 3 to end: 2
You entered 2 so function3 was called

Enter a number between 0 and 2, 3 to end: 3
You entered 3 to end
```

Рис. 7.28. Пример массива указателей на функцию (часть 2 из 2)

В этом вызове **f[choice]** выбирает указатель по индексу **choice**. Для вызова функции указатель разыменовывается, а переменная **choice** передается функции в качестве аргумента. Каждая функция выводит значение полученного аргумента и свое имя, что позволяет убедиться в правильности вызова функции. В упражнениях вы разработаете программу, управляемую с помощью меню.

Резюме

- Указатели являются переменными, которые в качестве значения содержат адреса других переменных.
- Указатели должны быть объявлены, прежде чем они будут использоваться.
- В строке

  ```
  int *ptr;
  ```

  переменная **ptr** объявляется указателем на объект целого типа, что читается как «**ptr** — указатель на целое». Символ * используется здесь как знак того, что переменная является указателем.
- Имеются три значения, которые могут использоваться для инициализации указателя: **0** (ноль), макрос **NULL** или адрес. Инициализации указателя значением **0** и **NULL** — идентичны.
- Единственное целое число, которое может быть присвоено указателю — это 0.
- Операция взятия адреса (&) возвращает адрес своего операнда.
- Операндом операции взятия адреса должна быть переменная; операция & не может применяться к константам, выражениям или к переменным, объявленным с модификатором **register**.
- Операция *, называемая операцией косвенной адресации или разыменования, возвращает значение объекта, на который ссылается ее операнд. Это называется разыменованием указателя.
- Когда функции передается аргумент, который она должна изменить, следует передавать его адрес. Для того, чтобы изменить значение аргумента в вызывающей функции, вызываемая функция должна использовать операцию разыменования (*).

- Аргументу, который передается по ссылке, должен соответствовать параметр-указатель.
- В прототипах функций должны быть обозначены типы указателей, при этом имена указателей можно не включать. Имена указателей могут быть включены в качестве комментариев, компилятор игнорирует их.
- При помощи модификатора **const** программист может сообщить компилятору, что значение переменной не должно изменяться.
- Компилятор перехватывает попытки изменения значения переменных, объявленных с модификатором **const** и выдает либо предупреждение, либо сообщение об ошибке, в зависимости от того, с каким компилятором вы работаете.
- Существуют четыре способа передачи указателя в функцию: не-константный указатель на не-константные данные, указатель-константа на не-константные данные, указатель — не-константа на данные-константы и указатель-константа на данные-константы.
- Массивы автоматически передаются по ссылке, потому что имя массива является указателем на начало массива.
- Для передачи по ссылке отдельного элемента массива нужно передавать его адрес.
- Во время компиляции программы можно определить размер массива (или любого другого типа данных) в байтах, используя унарную операцию **sizeof**.
- Операция **sizeof**, примененная к имени массива, возвращает общее количество байтов в массиве как целое число.
- Операция **sizeof** может применяться к переменной любого типа или к константе.
- С указателями могут выполняться следующие арифметические операции: инкремент (++) указателя, декремент (--) указателя, сложение (+ или +=) указателя и целого числа, вычитание (- или -=) из указателя целого числа и вычитание одного указателя из другого.
- При прибавлении или вычитании целого числа из указателя последний увеличивается или уменьшается на произведение этого числа и размера объекта, на который ссылается указатель.
- Арифметические операции над указателем следует применять при работе с «непрерывными» областями памяти, например, с массивом. Все элементы массива располагаются в памяти последовательно, друг за другом.
- Результаты арифметических операций над указателем на символьный массив будут совпадать с результатами традиционной арифметики, потому что для хранения одного символа отводится один байт памяти.
- Указатели могут присваиваться друг другу, если оба указателя имеют один и тот же тип. В противном случае нужно выполнять операцию приведения типа. Исключением из этого правила является указатель на **void**, являющийся обобщенным указателем, который может ссылаться на данные любого типа. Указателю на **void** можно присвоить значение указателя любого типа, указателю любого типа можно присвоить значение **void**-указателя без приведения типа.

- Указатель типа **void *** не может разыменовываться.
- К указателям можно применять операции равенства и сравнения. Сравнение указателей обычно имеет смысл, если указатели ссылаются на элементы одного и того же массива.
- С указателями, как и с именами массива, можно использовать индексы.
- Имя массива без индекса является указателем на первый элемент массива.
- В нотации указатель/смещение смещение имеет тот же смысл, что и индекс массива.
- Любые выражения с индексацией массива могут быть записаны с помощью указателя и смещения, где в качестве указателя может использоваться либо имя массива, либо иной указатель, ссылающийся на начало массива.
- Имя массива является указателем-константой на фиксированное место в памяти. В отличие от обычных указателей его значение не может изменяться.
- Указатели можно объединять в массивы.
- Указатели могут ссылаться на функции.
- Указатель на функцию ссылается на начальный адрес программного кода функции.
- Указатели на функции могут быть переданы функциям, возвращаться функциями, сохраняться в массивах и присваиваться другим указателям.
- Часто указатели на функции находят применение в системах, управляемых с помощью меню.

Терминология

const
sizeof
void * (указатель на **void**)
арифметика указателей
бесконечня отсрочка
вызов по значению
вызов по ссылке
выражение с указателями
вычитание двух указателей
вычитание целого из указателя
генерация вызова по ссылке
декремент указателя
динамическое распределение памяти
индексация указателя
инициализация указателей
инкремент указателя
константный указатель
константный указатель на константные данные
константный указатель на не-константные данные
косвенная адресация
косвенная ссылка на переменную
массив строк
массив указателей
не-константный указатель на не-константные данные
не-константный указатель на константные данные
нисходящее пошаговое уточнение
нотация указатель/смещение
операция взятия адреса
операция косвенной адресации (*)
операция разыменования (*)
принцип минимума привилегий
присваивание указателя
прямая ссылка на переменную
разыменование указателя
связанный список

сложение указателя с целым
смещение
сравнение указателей
типы указателей
указатель
указатель NULL
указатель на void (void *)
указатель на символ
указатель на функцию

Распространенные ошибки программирования

7.1. Знак операции косвенной адресации * не распространяется на все переменные в строке объявления. Символ * должен предшествовать имени каждого указателя.

7.2. Разыменование указателя, который не был инициализирован или которому не присвоили необходимое значение адреса. Это приводит к фатальной ошибке во время выполнения программы или случайной порче данных, в результате чего выполнение программы завершается с неверным результатом.

7.3. Не разыменовывайте указатель, кроме случая, когда вам нужно получить значение, на которое он ссылается.

7.4. Функции ошибочно передается параметр по значению, а не по ссылке. В таких случаях некоторые компиляторы поступят с полученным аргументом как с указателем и произведут разыменование этого «указателя». В результате во время выполнения программы произойдет ошибка обращения к памяти или ошибка сегментации. Компиляторы, проверяющие аргументы и параметры на соответствие типов, выдадут сообщение об ошибке.

7.5. Использование арифметических операций с указателем, ссылающимся не на элементы массива.

7.6. Вычитание или сравнение двух указателей, ссылающихся не на один и тот же массив.

7.7. Выход за начало или конец массива при арифметических операциях с указателем.

7.8. Присвоение значения указателя одного типа указателю другого типа, когда ни один из них не является указателем типа **void** *, приводит к синтаксической ошибке.

7.9. Разыменование **void**-указателя.

7.10 Попытка изменения имени массива, являющегося указателем-константой, приводит к синтаксической ошибке.

Хороший стиль программирования

7.1. Включайте в имена переменных-указателей префикс (или суффикс) **ptr**; такие имена подскажут вам, что эти переменные являются указателями и должны обрабатываться соответствующим образом.

7.2. Чтобы избежать неожиданных результатов, всегда инициализируйте указатели.

7.3. Используйте вызов по ссылке только в том случае, когда вызываемая функция должна изменять передаваемый ей аргумент. (Еще один пример принципа минимума привилегий.)

7.4. Прежде чем использовать функцию, посмотрите на ее прототип, чтобы узнать, может ли функция изменять переданные ей значения.

7.5. Ссылайтесь на элементы массива при помощи индексов. Пусть ваша программа будет компилироваться немного дольше, но зато текст программы будет более понятен.

Советы по повышению эффективности

7.1. При передаче больших объектов данных, например, структур, используйте указатель на константные данные. Этот подход позволит вам совместить производительность передачи параметра по ссылке и защиту данных как при передаче параметра по значению.

7.2. Передача параметра требует времени и места в стеке для создания копии передаваемой функции. Глобальным переменным не требуется этих дополнительных ресурсов, так как они доступны в любом месте программы.

7.3. Ссылка на элемент массива при помощи индекса преобразуется во время компиляции в выражение с указателем, поэтому, если вы сами будете обращаться к элементам массива при помощи указателей, то сократите время компиляции вашей программы.

7.4. Часто «естественный» алгоритм может столкнуться с неожиданными препятствиями при своем выполнении, как, например, бесконечная отсрочка. Ищите алгоритмы, которые не содержат в себе таких «бесконечностей».

Советы по переносимости программ

7.1. Хотя модификатор const определен в стандарте ANSI C, некоторые системы его не поддерживают.

7.2. Размер в байтах одного и того же типа данных может быть разным в различных системах. При написании программ, которые будут работать на различных платформах и при этом небезразличны к представлению данных, используйте операцию sizeof для определения размера типа данных.

7.3. Большинство компьютеров сегодня поддерживают 2-байтовые и 4-байтовые целые. Некоторые из последних машин имеют еще и 8-байтовые целые. Поскольку результат арифметики указателей зависит от размера объектов, на которые ссылается указатель, то результат арифметических выражений с указателями является машинно-зависимым.

Общие методические замечания

7.1. Употребление модификатора **const** можно рассматривать в рамках соблюдения принципа минимальных привилегий. Следование этому

принципу позволяет существенно сократить время отладки программы и количество побочных нежелательных эффектов, а также делает программу проще для понимания и сопровождения.

7.2. Если передаваемое функции значение не изменяется (или не должно быть изменено) в теле функции, оно должно объявляться с модификатором **const**, чтобы гарантировать невозможность даже случайного ее изменения.

7.3. В результате вызова по значению в вызывающей функции может быть изменена только одна величина. Этой величине должно быть присвоено возвращаемое вызванной функцией значение. Для изменения сразу нескольких значений в вызывающей функции необходимо использовать вызов по ссылке.

7.4. Размещение прототипа функции в теле другой функции ограничивает корректные вызовы первой функции только теми, которые будут сделаны из функции, где объявлен прототип. Это опять принцип минимума привилегий.

7.5. При передаче в функцию массива следует передавать и его размер. Это сделает функцию более универсальной. Универсальные функции могут использоваться большим количеством программ.

7.6. Глобальные переменные нарушают принцип минимальных привилегий и являются примером плохого стиля программирования.

Упражнения для самоконтроля

7.1. Заполните пропуски в предложениях:

a) Указатель — это переменная, которая в качестве значения содержит _____ другой переменной.

b) Только три величины могут использоваться для инициализации указателя: _____, _____ или _____.

c) Единственное целое число, которое может быть присвоено указателю, это _____.

7.2. Являются ли следующие утверждения верными? Если утверждение неверно, объясните, почему.

a) Операция взятия адреса & может применяться только к константам, выражениям и переменным, объявленным с модификатором **register**.

b) Указатель на **void** может быть разыменован.

c) Указатели на разные типы данных не могут быть присвоены друг другу без использования операции приведения типов.

7.3. Выполните следующие задания. Предположим, что числа с плавающей точкой обычной точности занимают 4 байта, и что начальный адрес массива равен 1002500. Учтите, что при выполнении очередного задания нужно использовать результаты предыдущих заданий (там, где это необходимо).

a) Объявите массив **numbers** типа **float** из 10 элементов и присвойте элементам значения **0.0**, **1.1**, **2.2**, ..., **9.9**. Используйте символическую константу **SIZE**, равную **10**.

b) Объявите указатель **nPtr**, ссылающийся на тип **float**.

c) Выведите элементы массива **numbers**, используя нотацию имя массива/индекс. Используйте цикл **for** и переменную цикла **i**. Выведите каждый элемент с точностью одного знака после запятой.

d) Присвойте указателю **nPtr** адрес начала массива **numbers** двумя способами.

e) Выведите элементы массива **numbers** при помощи обращения к элементам массива по методу указатель/смещение, где в качестве указателя используется **nPtr**.

f) Выведите элементы массива **numbers** при помощи обращения к элементам массива по методу указатель/смещение, где в качестве указателя используется имя массива.

g) Выведите элементы массива **numbers**, используя индексацию указателя **nPtr**.

h) Сошлитесь на четвертый элемент массива, используя все четыре способа доступа к элементам массива: имя массива/индекс, имя массива/смещение, указатель/индекс с указателем **nPtr** и указатель/смещение с указателем **nPtr**.

i) Если предположить, что **nPtr** указывает на начало массива numbers, то на какой адрес ссылается **nPtr + 8**? Что за значение находится по указанному адресу?

j) Предположим, что **nPtr** ссылается на **numbers[5]**; на какой адрес ссылается **nPtr -= 4**? Какое значение находится по этому адресу?

7.4. Выполните каждую из следующих задач при помощи одиночного оператора. Считайте объявленными переменные с плавающей точкой **number1** и **number2**, причем **number1** присвоено значение 7.3.

a) Объявите указатель **fPtr** на тип **float**.

b) Присвойте адрес переменной **number1** указателю **fPtr**.

c) Выведите значение величины, на которую ссылается **fPtr**.

d) Присвойте значение величины, на которую указывает **fPtr**, переменной **number2**.

e) Выведите значение **number2**.

f) Выведите адрес **number1**. Используйте спецификатор формата **%p**.

g) Выведите адрес, сохраненный в **fPtr**. Используйте спецификатор формата **%p**. Совпало ли выведенное значение адреса с адресом **number1**?

7.5. Выполните следующие задания.

a) Напишите заголовок функции **exchange**, которая имеет два параметра-указателя на переменные с плавающей точкой **x** и **y** и не возвращает значения.

b) Напишите прототип для функции из предыдущего задания.

c) Напишите заголовок функции **evaluate**, которая возвращает целое число и имеет в качестве параметров целое число **x** и указатель на функцию **poly**. Функция **poly**, в свою очередь, имеет целочисленный параметр и возвращает целое число.

d) Напишите прототип для функции из предыдущего задания.

7.6. Найдите ошибку в каждом из следующих фрагментов программ. Учитывайте следующую информацию:

```
int *zPtr; /* zPtr ссылается на массив z */
int *aPtr = NULL;
void *sPtr = NULL;
int number, I;
int z[5] = {1, 2, 3, 4, 5};

sPtr = z;
```

a) `++zptr;`

b)
```
/* использование указателя для получения первого элемента
   массива */
number = zPtr;
```

c)
```
/* присвоение переменной number значения второго элемента
массива (число 3)*/
number = *zPtr[2];
```

d)
```
/* вывод элементов массива z */
for (i = 0; i <= 5; I++)
   printf("%d ", zPtr[i]);
```

e)
```
/* присвоение переменной number значения, на которое
   ссылается указатель sPtr  */
number = *sPtr;
```

f) `++z;`

Ответы на упражнения для самоконтроля

7.1. a) адрес. b) **0**, **NULL**, адрес. c) **0**.

7.2. a) Неверно. Операция взятия адреса может применяться только к переменным, причем только к тем, при объявлении которых не использовался модификатор **register**.

b) Неверно. Указатель типа **void** не может быть разыменован, потому что неизвестно, сколько байт памяти должно занимать значение, полученное при разыменовании.

c) Неверно. Указатели типа **void** могут быть присвоены указателям других типов, и **void**-указателям можно присваивать значения указателей других типов.

7.3. a)
```
float numbers[SIZE] = {0.0, 1.1, 2.2, 3.3, 4.4, 5.5,
                       6.6, 7.7, 8.8, 9.9};
```

b) `float *nPtr;`

c)
```
for (i = 0; i <= SIZE - 1; I++)
   printf("%.1f" , numbers[i]);
```

d)
```
nPtr = numbers;
nPtr = &numbers[0];
```

e)
```
for (i = 0; i <= SIZE - 1; I++)
   printf("%.1f  ", *(nPtr + i));
```

f)
```
for (i = 0; i <= SIZE - 1; I++)
   printf("%.1f  ", *(numbers + i));
```

g)
```
for (i = 0; i <= SIZE - 1; I++)
   printf("%.1f  ", nPtr[i]);
```

h)
```
numbers[4]
*(numbers + 4)
nPtr[4]
*(nPtr + 4)
```

i) Адрес равен **1002500 + 8 * 4 = 1002532.** По этому адресу находится значение **8. 8.**

j) Адрес элемента **numbers[5]** равен **1002500 + 5 * 4 = 1002520.**

В результате выполнения операции **nPtr -= 4** адрес в **nPtr 1002520 - 4 * 4 = 1002504.**

По этому адресу находится значение **1.1.**

7.4. a)
```
float *fPtr;
```

b)
```
fPtr = &number1;
```

c)
```
printf("The value of *fPtr is %f\n", *fPtr);
```

d)
```
number2 = *fPtr;
```

e)
```
printf("The value of number2 is %f\n", number2);
```

f)
```
printf("The address of number1 is %p\n", &number1);
```

g)
```
printf("The address stored in fPtr is %p\n", fPtr);
```

Да, эти значения совпадают.

7.5. a)
```
void exchange(float *x, float *y)
```

b)
```
void exchange(float *, float *);
```

c)
```
int evaluate(int x, int (*poly)(int))
```

d)
```
int evaluate(int, int (*)(int));
```

7.6. a) Ошибка: **zPtr** не был инициализирован.

Исправление: инициализируйте **zPtr** при помощи следующего выражения **zPtr = z;**

b) Ошибка: указатель не был разыменован.

Исправление: воспользуйтесь оператором следующего вида: **number = *zPtr;**

c) Ошибка: **zPtr[2]** не является указателем и не должен разыменовываться.

Исправление: замените ***zPtr[2]** на **zPtr[2]**.

d) Ошибка: ссылка на элемент массива со значением индекса, которое вышло за границы массива.

Исправление: замените конечное значение переменной цикла **for** на значение **4.**

e) Ошибка: разыменование **void**-указателя.

Исправление: чтобы разыменовать этот указатель, нужно сначала привести его к виду указателя на целое. Примените оператор вида **number = * (int *) sPtr;**

f) Ошибка: попытка изменить значение указателя — имени массива — в арифметическом выражении.

Исправление: используйте в арифметическом выражении вместо имени массива переменную-указатель, или индекс и имя массива, для обращения к нужному элементу.

Упражнения

7.7. Заполните пропуски в предложениях:

a) Операция _____ возвращает адрес своего операнда.

b) Операция _____ возвращает значение объекта, на который операнд указывает.

c) Для того, чтобы произвести вызов по ссылке при передаче переменной (не массива) в качестве аргумента функции, нужно передать _____ переменной.

7.8. Являются ли следующие утверждения верными? Если утверждение неверно, объясните, почему.

a) Не имеет смысла сравнивать два указателя, которые указывают на различные массивы.

b) Поскольку имя массива является указателем на первый элемент массива, имена массивов могут использоваться совершенно аналогично указателям.

7.9. Выполните следующие задания. Считайте, что целые числа без знака занимают в памяти 2 байта, и что начальный адрес массива равен 1002500.

a) Объявите массив типа целое без знака с именем **values**, состоящий из 5 элементов, и инициализируйте элементы массива четными целыми числами от 2 до 10. Используйте символическую константу **SIZE**, равную **5**.

b) Объявите указатель **vPtr** на объект типа **unsigned int**.

c) Выведите элементы массива **values** с использованием обращения к элементам массива по методу имя массива/индекс. Используйте для этой цели цикл **for** с управляющей переменной **i**, которую считайте объявленной.

d) Запишите два оператора, при помощи которых можно присвоить начальный адрес массива **values** указателю **vPtr**.

e) Выведите элементы массива **values**, используя нотацию указатель/смещение.

f) Выведите элементы массива **values**, используя нотацию указатель/смещение, где в качестве указателя используется имя массива.

g) Выведите элементы массива **values**, используя индекс с указателем на массив.

h) Сошлитесь на 5-й элемент массива **values**, используя индексацию массива, нотацию указатель/смещение с именем массива в качестве указателя, индексацию указателя и нотацию указатель/смещение.

i) На какой адрес ссылается выражение **vPtr + 3**? Чему равно находящееся по этому адресу значение?

j) Если предположить, что **vPtr** ссылается на **values[4]**, чему будет равно значение адреса, находящегося в **vPtr**, после выполнения оператора **vPtr -= 4**? Какое значение находится по данному адресу?

7.10. Выполните каждое из следующих заданий, используя для этой цели только один оператор. Считайте, что переменные **value1** и **value2** типа **long** объявлены и переменной **value1** присвоено значение **200000**.

a) Объявите указатель **lPtr** на объект данных типа **long**.

b) Присвойте значение адреса переменной **value1** указателю **lPtr**.

c) Выведите значение объекта, на который ссылается указатель **lPtr**.

d) Присвойте значение объекта, на который ссылается **lPtr**, переменной **value2**.

e) Выведите значение **value2**.

f) Выведите адрес **value1**.

g) Выведите значение адреса, находящееся в **lPtr**. Совпадает ли выведенное значение с адресом **value1**?

7.11. Выполните следующие задания.

a) Напишите заголовок функции **zero**, которая имеет параметром целочисленный массив **bigIntegers** типа **long** и не возвращает значение.

b) Напишите прототип для функции из предыдущего задания.

c) Напишите заголовок функции **addAndSum**, имеющей целочисленный массив **oneTooSmall** в качестве параметра и возвращающей целое число.

d) Напишите прототип функции из предыдущего задания.

Замечание: Упражнения с 7.12 по 7.15 являются достаточно сложными. Но если вы с ними справитесь, то сможете легко программировать популярные карточные игры.

7.12. Измените программу на рис. 7.24 так, чтобы функция раздачи сдавала по пять карт, необходимых для игры в покер. Затем напишите следующие дополнительные функции, которые могут:

a) Определить, имеется ли на руках у игрока пара.

b) Определить, имеется ли на руках у игрока две пары.

c) Определить, имеется ли на руках тройка (например, три вальта).

d) Определить, имеет ли игрок каре (например, четыре туза).

e) Определить, имеется ли на руках флеш (т.е. пять карт одной масти).

f) Определить, имеется ли на руках стрит (т.е. пять карт последовательных номиналов).

7.13. Используя функции, разработанные в упражнении 7.12, напишите программу, которая сдает двум игрокам в покер на руки по пять карт и оценивает, чья карта лучше.

7.14. Измените программу, разработанную в упражнении 7.13 таким образом, чтобы она исполняла роль игрока, сдающего карты. Карты сдающего кладутся «лицом вниз», так что играющий с программой игрок их не видит. Программа должна затем оценить карту сдающего и, основываясь на качестве карт, сдающий должен взять себе одну, две или три карты взамен не устраивающих его карт из первоначально розданных. После этого программа должна оценить карты сдающего еще раз. (Предостережение: это — трудная задача!)

7.15. Измените программу, разработанную в упражнении 7.14 так, чтобы она манипулировала картами сдающего, а играющий с программой решал бы сам, какие карты ему нужно менять. Программа должна затем оценивать карты играющих и определять победителя. Теперь сыграйте с этой программой 20 игр. Кто выиграет большее количество игр, вы или компьютер? После этого пусть один из ваших друзей сыграет еще 20 игр с компьютером. Кто выиграет большее количество игр? Основываясь на результатах этих игр, сделайте соответствующие изменения, которые позволили бы усовершенствовать вашу программу игры в покер (это также будет трудной задачей). Сыграйте еще 20 игр. Не стала ли ваша программа играть лучше?

7.16. В программе тасовки и сдачи карт, текст которой приведен на рис. 7.24, мы преднамеренно использовали неэффективный алгоритм тасования, который может привести к нескончаемым отсрочкам. В связи с этой проблемой вам предстоит создать быстродействующий алгоритм тасования, в котором этой опасности просто не существует.

Измените программу на рис. 7.24 следующим образом. Начните с инициализации массива **deck** так, как показано на рис. 7.29. Измените функцию **shuffle** таким образом, чтобы она в цикле по строкам и столбцам массива перебирала элементы массива по разу и меняла местами значения текущего элемента и случайно выбранного элемента массива.

	0	1	2	3	4	5	6	7	8	9	10	11	12
0	1	2	3	4	5	6	7	8	9	10	11	12	13
1	14	15	16	17	18	19	20	21	22	23	24	25	26
2	27	28	29	30	31	32	33	34	35	36	37	38	39
3	40	41	42	43	44	45	46	47	48	49	50	51	52

Рис. 7.29. Неперетасованный массив **deck**

Выведите полученный в результате массив, чтобы определить, достаточно ли хорошо перетасована колода (как, например, на рис. 7.30). Вы можете вызывать в вашей программе функцию **shuffle** несколько раз, чтобы гарантировать «хорошую» перетасовку.

	0	1	2	3	4	5	6	7	8	9	10	11	12
0	19	40	27	25	36	46	10	34	35	41	18	2	44
1	13	28	14	16	21	30	8	11	31	17	24	7	1
2	12	33	15	42	43	23	45	3	29	32	4	47	26
3	50	38	52	39	48	51	9	5	37	49	22	6	20

Рис. 7.30. Пример перетасованного массива **deck**

Заметим теперь, что хотя мы и улучшили алгоритм тасования, сдающий алгоритм все равно имеет недостаток: он ищет в массиве **deck** карту с номером 1, затем карту с номером 2, затем карту номер 3 и т.д. Положение ухудшается еще и тем, что после того, как алгоритм раздачи нашел и сдал карту, он продолжает поиск уже сданной карты в оставшейся части массива **deck**. Измените программу из рис. 7.24 так, чтобы поиск прекращался, как только найдена нужная карта, и после этого начинался процесс сдачи следующей карты. В главе 10 мы разработаем эффективный алгоритм сдачи, который требует выполнения только одной операции на карту.

7.17. (*Компьютерное моделирование: Заяц и Черепаха*) В этом упражнении вам предстоит воспроизвести одно из великих исторических событий, а именно, классический забег черепахи и зайца. Вам нужно будет использовать генератор случайных чисел, чтобы разработать программу, моделирующую это незабываемое соревнование.

Наши соперники начинают гонку в «квадрате 1» на дистанции, состоящей из 70 квадратов. Каждый квадрат представляет из себя возможную позицию на трассе гонки. Финишная линия — это квадрат 70. Спортсмен, который достигнет или проскочит квадрат 70 первым, вознаграждается ведром свежей моркови и салата. Трасса забега проходит по склону скользкой горы, так что иногда соперники поскальзываются и скатываются назад.

Движение соперников регулируется часами, делающими один отсчет в секунду. Ежесекундно ваша программа должна корректировать позиции животных на трассе согласно следующим правилам:

Животное	Тип движения	Процент времени	Описание движения
Черепаха	Тащится быстро	50%	3 квадрата вправо
	Сползание	20%	6 квадратов влево
	Тащится медленно	30%	1 квадрат вправо
Заяц	Спячка	20%	Движения нет
	Большой прыжок	20%	9 квадратов вправо
	Большое сползание	10%	12 квадратов влево
	Малый прыжок	30%	1 квадрат вправо
	Малое сползание	20%	2 квадрата влево

Используйте переменные для хранения позиции животного на трассе (т.е. числа величиной от 1 до 70). Каждое животное начинает свое движение с позиции 1. Если животное соскальзывает за позицию 1, передвигайте его в начальную позицию — квадрат 1.

Выбирайте тип движения в соответствии с частотными процентами, приведенными в таблице, разыгрывая случайное целое число, i, в диапазоне 1 <= i <= 10. Для черепахи, «тащится быстро» выпадает в случае, когда 1 <= i <= 5, «сползание» — когда 6 <= i <= 7 и «тащится медленно» — когда 8 <= i <= 10. Используйте этот же метод для выбора движения зайца.

Начинайте забег с вывода сообщения

BANG !!!!!

AND THE'RE OFF !!!!!

Затем, с каждым шагом часов (т.е. с каждым шагом цикла) выводите строку из 70-ти позиций, показывая символ **T (Tortoise)** в позиции черепахи и символ **H (Hare)** — в позиции зайца. Часто соперники будут попадать в один и тот же квадрат. В этом случае черепаха будет кусать зайца, а ваша программа должна выводить сообщение **«OUCH!!!»**, начиная с этого квадрата. Все остальные позиции, кроме тех, где выведено **T**, **H** или **OUCH!!!**, должны быть пустыми.

После вывода очередной строки проверяйте, не удалось ли кому-либо из животных достичь или перепрыгнуть квадрат с номером 70. Если это так, то выводите имя победителя и заканчивайте процесс моделирования. Если победила черепаха, выводите строку **«TORTOISE WINS!!! YAY!!!»** Если выиграл заяц, то строка должна быть такой: **«Hare wins. Yuch»**. Если оба животных приходят к финишу одновременно, вы можете отдать предпочтение черепахе («аутсайдеру») или вывести сообщение о том, что забег закончился вничью. Если на данном шаге никто не достиг финиша, начинайте новый шаг цикла, соответствующий следующему временному отсчету. Когда вы готовы будете выполнить вашу программу, пригласите группу болельщиков, чтобы наблюдать за соревнованиями. Вы удивитесь тому, как ваша аудитория будет увлечена происходящим!

Специальный раздел: как самому построить компьютер

В следующих задачах мы временно отойдем далеко от программирования на языке высокого уровня. Мы «снимем корпус» компьютера и рассмотрим его внутреннее устройство. Мы познакомим вас с программированием на машинном языке и напишем на нем несколько программ. И, чтобы сделать этот опыт особенно ценным, мы затем «построим» компьютер (методом программного *моделирования*), на котором вы сможете выполнять ваши программы на машинном языке!

7.18. (*Программирование на машинном языке*) Мы создадим компьютер и назовем его Симплетрон. Как следует из самого имени компьютера, это — простая машина (от англ. simple — простой), но, как мы скоро в этом убедимся, машина мощная. Симплетрон выполняет програм-

мы, написанные на единственном языке, который он способен понимать и который мы назовем Машинным языком Симплетрона (Simpletron Mashine Language), или сокращенно SML.

Симплетрон имеет *аккумулятор* — «специальный регистр», в который помещается информация для выполнения вычислений или различных операций сравнения. Вся информация в Симплетроне хранится в виде *слов*. Словом будем здесь называть десятичное число из четырех цифр со знаком, например, **+3364**, **-1293**, **+0007**, **-0001** и т.д. Симплетрон имеет память размером в 100 слов, ссылка на которые производится посредством чисел **00**, **01**, ..., **99**.

Перед тем как запустить программу на SML, мы должны *загрузить* эту программу в память. Первая команда любой программы на SML всегда помещается по адресу **00**.

Каждая команда SML занимает одно слово памяти Симплетрона (и, следовательно, команды являются десятичными числами из четырех цифр со знаком). Мы договоримся о том, что команды SML всегда имеют знак плюс, тогда как знак слова данных может быть либо плюсом, либо минусом. Каждое слово в памяти Симплетрона может содержать либо команду, либо данные, обрабатываемые программой, или может быть неиспользуемым (в этом случае значение его не определено). Первые две цифры каждой SML-команды — это *код операции*, которую нужно выполнить. Допустимые коды операций SML представлены на рис. 7.31.

Код операции	Значение
Операции ввода/вывода	
#define READ 10	Вводит слово с терминала в указанное место памяти
#define WRITE 11	Выводит на терминал слово из указанного адреса памяти
Операции загрузки/выгрузки	
#define LOAD 20	Помещает в аккумулятор слово из указанного адреса памяти
#define STORE 21	Выгружает слово из аккумулятора по указанному адресу памяти
Арифметические операции	
#define ADD 30	Выполняет сложение слова в аккумуляторе и слова из указанного места в памяти (результат операции остается в аккумуляторе)
#define SUBTRACT 31	Вычитает из слова в аккумуляторе слово из указанного места в памяти (результат операции остается в аккумуляторе)
#define DIVIDE 32	Выполняет деление слова в аккумуляторе на слово из указанного места в памяти (результат операции остается в аккумуляторе)
#define MULTIPLY 33	Вычисляет произведение слова в аккумуляторе и слова из указанного места в памяти (результат операции остается в аккумуляторе)
Операции передачи управления	
#define BRANCH 40	Переход к указанному адресу памяти

Код операции	Значение
#define BRANCHNEG 41	Переход к указанному адресу памяти, если в аккумуляторе находится отрицательное число
#define BRANCHZERO 42	Переход к указанному адресу памяти, если в аккумуляторе находится ноль
#define HALT 43	Останов, выполняется при завершении программой своей работы

Рис. 7.31. Коды операций SML – Машинного языка Симплетрона

Последние две цифры SML-команды — *операнд*, являющийся адресом слова, над которым выполняется операция. А теперь давайте разберем несколько простых программ на SML.

Пример 1 Адрес	Слово	Описание
00	+1007	(Ввод A)
01	+1008	(Ввод B)
02	+2007	(Загрузка A в аккумулятор)
03	+3008	(Прибавить B)
04	+2109	(Выгрузить в C)
05	+1109	(Вывод C)
06	+4300	(Останов)
07	+0000	(Переменная A)
08	+0000	(Переменная B)
09	+0000	(Результат C)

Эта программа на SML считывает два числа, введенных с клавиатуры, вычисляет их сумму и выводит ее. Команда **+1007** вводит первое число и помещает его в слово с адресом **07** (которое было инициализировано нулем). Затем команда **+1008** вводит следующее число и записывает его по адресу **08**. Команда загрузки, **+2007**, помещает первое число в аккумулятор, а команда сложения, **+3008**, прибавляет второе число к числу, записанному в аккумуляторе. Все арифметические команды SML оставляют результат вычислений в аккумуляторе. Команда выгрузки, **+2109**, помещает результат, находящийся в аккумуляторе, обратно в память по адресу **09**, из которого команда вывода, **+1109**, выводит это число (в виде десятичного целого числа из четырех цифр со знаком). Команда останова, **+4300**, завершает выполнение программы.

Следующая SML-программа вводит с клавиатуры два числа, определяет большее из них и выводит это значение. Обратите внимание на команду **+4107** — условную передачу управления, напоминающую оператор **if** в языке C. Теперь сами напишите программы на SML, выполняющие следующие задачи.

Пример 2 Адрес	Слово	Описание
00	+1009	(Ввод A)
01	+1010	(Ввод B)
02	+2009	(Загрузка A в аккумулятор)
03	+3110	(Отнять B)
04	+4107	(Переход на 07, если отрицательное)
05	+1109	(Вывод A)
06	+4300	(Останов)
07	+1110	(Вывод B)
08	+4300	(Останов)
09	+0000	(Переменная A)
10	+0000	(Переменная B)

a) Напишите цикл, прерываемый вводом контрольного значения, для ввода 10 положительных чисел, вычисления суммы этих чисел и ее вывода.

b) Используя управляемый счетчиком цикл, введите семь чисел, положительных и отрицательных, а затем вычислите и выведите их среднее значение.

c) Введите ряд чисел, определите большее среди них и выведите найденное значение. В первом вводимом числе указывается количество чисел, которые должны быть обработаны.

7.19. (*Программная модель компьютера*) Вам это может показаться возмутительным, но для решения этой задачи вам нужно собрать свой собственный компьютер. Нет, вам не придется брать в руки паяльник и отвертку. Вместо этого вы воспользуетесь мощным методом компьютерного моделирования и создадите программную модель (*симулятор*) Симплетрона. Вы не будете разочарованы. Программа-симулятор Симплетрона превратит ваш компьютер в «настоящий» Симплетрон и вы сможете исполнять, тестировать и отлаживать программы SML, которые вы написали в упражнении 7.18.

При запуске вашего Симплетрона на экран должен выводиться примерно такой текст:

```
*** Симплетрон приветствует вас! ***
*** Пожалуйста, введите вашу программу, по одной команде  ***
*** (или слову данных) за раз. Я буду выводить в качестве ***
*** подсказки текущий адрес и знак вопроса (?). Введенное ***
*** вами слово будет размещено по указанному адресу. Для  ***
*** прекращения ввода программы введите число  -99999.     ***
```

В качестве модели памяти Симплетрона объявите одномерный массив **memory** из 100 элементов. Теперь представим, что наша симулятор запущен, и разберем процесс ввода программы примера 2 из упражнения 7.18:

```
00? +1009
01? +1010
02? +2009
03? +3110
04? +4107
05? +1109
06? +4300
07? +1110
08? +4300
09? +0000
10? +0000
11? -99999
*** Загрузка программы завершена ***
*** Начинаю выполнение программы ***
```

Программа SML теперь помещена (т. е. загружена) в массив **memory**. Теперь Симплетрон должен выполнить вашу программу. Выполнение программы начинается с команды, расположенной по адресу 00 и, подобно C, продолжается последовательно, если не встретится команда передачи управления в другую часть программы.

Для представления аккумулятора используйте переменную **accumulator**. В переменной **instructionCounter** храните адрес следующей исполняемой команды. Для хранения кода операции текущей исполняемой команды, т.е. левых двух цифр слова команды, используйте переменную **operationCode**. В переменной **operand** можно хранить адрес операнда текущей исполняемой команды. Напомним, что адрес операнда — это две правые цифры командного слова. Не выполняйте команды непосредственно из памяти. Используйте промежуточную переменную **instructionRegister** для хранения исполняемой команды. И уже из нее выделите две левые цифры кода операции, поместите их в переменную **operationCode**, затем выделите правые две цифры адреса операнда команды и поместите их в переменную **operand**.

Перед тем, как Симплетрон начнет выполнение программы, наши переменные-регистры должны быть инициализированы следующим образом:

```
accumulator          +0000
instructionCounter     00
instructionRegister  +0000
operationCode          00
operand                00
```

Теперь давайте «пройдем» выполнение первой SML-команды, **+1009**, расположенной по адресу **00**. Процесс выполнения команды называется *циклом исполнения команды*.

В регистре команд **instructionCounter** расположен адрес следующей исполняемой команды. Мы выбираем слово этой команды из массива **memory** с помощью следующего оператора:

```
instructionRegister = memory[instructionCounter];
```

Извлечем из регистра команд код операции и адрес операнда, используя операторы

```
operationCode = instructionRegister / 100;
operand = instructionRegister % 100;
```

Теперь Симплетрон должен определить, что данному коду операции соответствует команда *ввода* (а не вывода, загрузки и т. д.). Нужный выбор из двенадцати возможных команд SML можно сделать в операторе **switch**.

В операторе **switch** различные SML-команды моделируются следующим образом (с остальными вы должны разобраться самостоятельно):

```
ввод:          scanf("%d", &memory[operand]);
загрузка:      accumulator = memory[operand];
сложение:      accumulator += memory[operand];
команды передачи управления: мы обсудим их чуть позже
останов:       эта команда выводит сообщение
          *** Симплетрон закончил свои вычисления ***
```

По окончании работы Симплетрон выводит имя и содержимое каждого регистра и своей памяти. Такая распечатка часто называется *компьютерным дампом*, или дампом памяти. Формат дампа показан на рис. 7.32. Он поможет вам в программировании функции, выводящей дамп памяти. Не забудьте, что после выполнения Симплетроном программы дамп памяти должен отражать фактические значения команд и данных на момент завершения программы.

Продолжим процесс разбора выполнения первой команды нашей программы, а именно **+1009**, расположенной по адресу **00**. Мы моделируем команду ввода в структуре **switch** при помощи оператора C

```
scanf("%d", &memory[operand]);
```

Перед выполнением функции **scanf** на экран должен быть выведен знак вопроса (**?**) в качестве запроса ввода данных. Симплетрон ждет до тех пор, пока пользователь не введет число и не нажмет клавишу Enter. Введенное значение будет помещено в ячейку памяти с адресом **09**.

На этом выполнение первой команды завершается. Остается только подготовить Симплетрон к выполнению следующей команды. Так как команда, которую мы только что выполнили, не является командой передачи управления, нам нужно просто увеличить регистр счетчика команд:

```
++instructionCounter;
```

На этом выполнение первой команды завершается полностью. После этого весь процесс (т.е. цикл исполнения команды) начинается снова с выборки из памяти следующей исполняемой команды.

Теперь давайте рассмотрим моделирование команд перехода (передачи управления). Все, что мы должны сделать — это правильно скорректировать значение регистра счетчика команд. Безусловная команда перехода (**40**) моделируется в операторе **switch** как

```
instructionCounter = operand;
```

```
РЕГИСТРЫ:
instructionCounter = operand;
accumulator          +0000
instructionCounter      00
```

```
instructionRegister  +0000
operationCode          00
operand                00

ПАМЯТЬ:
       0     1     2     3     4     5     6     7     8     9
 0 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
10 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
20 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
30 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
40 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
50 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
60 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
70 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
80 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
90 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
```

Рис. 7.32. Пример дампа памяти

Условная команда перехода «переход, если аккумулятор — ноль» может быть реализована при помощи следующих операторов:

```
if (accumulator == 0)
   instructionCounter = operand;
```

Теперь можете приступать к написанию программы — модели Симплетрона, через которую вы должны пропустить все программы, разработанные в упражнении 7.18. Вы можете добавить в SML какие-либо усовершенствования, но не забудьте включить их и в модель Симплетрона.

Ваш симулятор должен уметь обрабатывать различные ошибки. Например, при вводе программы входные данные должны находится в интервале от **-9999** до **+9999**. Поэтому ввод данных должен происходить внутри цикла **while**, в котором производится проверка введенной величины и, если введено неверное значение, процесс ввода должен продолжаться до тех пор, пока пользователь не введет верное значение.

Во время выполнения программы симулятор должен обрабатывать различные серьезные ошибки, например, попытку деления на ноль, попытку выполнить команду с неправильным кодом операции, переполнение аккумулятора (т.е. получение в процессе вычислений величины большей **+9999** или меньшей **-9999**) и другие. Такие серьезные ошибки называются *фатальными ошибками*. В случае возникновения такой ошибки симулятор должен выдать сообщение вроде

```
*** Попытка деления на ноль ***
*** Симплетрон аварийно завершил выполнение программы ***
```

и выдать полный дамп памяти в формате, предложенном выше. Этот дамп может помочь пользователю обнаружить ошибку в программе.

7.20. Измените программу тасования и сдачи карт из рис. 7.24 так, чтобы процесс тасования и сдачи выполняла бы одна функция (**shuffleAndDeal**). Подобно функции **shuffle** из рис. 7.24 эта функция должна содержать один вложенный цикл.

7.21. Что делает эта программа?

```
#include <stdio.h>

void mystery1(char *, const char *);

main()
{
   char string1[80], string2[80];

   printf("Enter two strings: ");
   scanf("%s%s"  string1, string2);
   mystery1(string1, string2);
   printf("%s\n", string1);
   return 0;
}

void mystery1(char *s1, const char *s2)
{
   while (*s1 != '\0')
      ++s1;

   for ( ; *s1 = *s2; s1++, s2++)
   ;  /* тело цикла пусто */
}
```

7.22. Что делает эта программа?

```
#include <stdio.h>

int mystery2(const char *);

main()
{
   char string[80];

   printf("Введите строку: ");
   scanf("%s", string);
   printf("%d\n", mystery2(string));
   return 0;
}

int mystery2(const char *s)
{
   int x = 0;

   for ( ; *s != '\0'; s++)
      ++x;

   return x;
}
```

7.23. Найдите ошибку в каждом из следующих фрагментов программы. Если ошибку можно исправить, то объясните как это можно сделать.

а)
```
int *number;
printf ("%d\n", *number);
```

b)
```
float *realPtr;
long *integerPtr;
integerPtr = realPtr;
```

c)
```
int *x, y;
x = y;
```

d)
```
char s[] =  "вот массив символов";
int count;
for ( ;  *s != '\0'; s++)
   printf ("%c ", *s);
```

e)
```
short *numPtr, result;
void *genericPtr = numPtr;
result = *genericPtr + 7;
```

f)
```
float x = 19.34;
float xPtr = &x;
printf( "%f\n" , xPtr);
```

g)
```
char *s;
printf( "%s\n", s);
```

7.24. (*Быстрая сортировка*) В примерах и упражнениях главы 6 мы обсуждали алгоритмы сортировки, такие, как пузырьковая или блочная сортировка. Мы представим вам теперь рекурсивный алгоритм сортировки, называемый быстрой сортировкой. Алгоритм для одномерного массива выглядит следующим образом:

1) *Этап разбиения:* Возьмите первый элемент несортированного массива и определите его расположение в отсортированном массиве. Это положение будет найдено, если все значения слева от данного элемента будут меньше, а все значения направо от элемента — больше значения данного элемента. Мы теперь имеем один элемент, расположенный на своем месте в отсортированном массиве и два несортированных подмассива.

2) *Этап рекурсии:* Выполните шаг 1 на каждом из несортированных подмассивов.

Каждый раз после выполнения шага 1 на подмассиве следующий элемент массива помещается на свое место в отсортированном массиве, и создаются два несортированных подмассива. Когда мы дойдем до подмассива, состоящего из одного элемента, этот элемент будет находится на своем окончательном месте в упорядоченном массиве.

Это описание алгоритма в целом кажется достаточно ясным, но как нам определить окончательную позицию первого элемента каждого подмассива? В качестве примера рассмотрим следующий набор значений (элемент, выделенный жирным — элемент разбиения, — должен быть помещен на свое окончательное место в отсортированном массиве):

37 2 6 4 89 8 10 12 68 45

1) Начиная с правого элемента массива будем сравнивать каждый элемент с числом **37** до тех пор, пока не будет найден элемент меньший чем **37**, после чего найденный элемент и **37** должны поменяться своими местами. Первым элементом, который меньше **37**, является число 12, поэтому они меняются местами. Теперь массив выглядит так:

12 2 6 4 89 8 10 **37** 68 45

Элемент 12 выделен курсивом, чтобы указать на то, что он поменялся местами с числом **37**.

2) Теперь начинаем движение с левой части массива, но начинаем со следующего элемента после 12, и сравниваем каждый элемент с **37**, пока не обнаружим элемент больший чем **37**, после чего меняем местами **37** и этот найденный элемент. В нашем случае первый элемент больший чем 37 — это 89, так что 37 и 89 меняются местами. Новый массив имеет вид:

12 2 6 4 **37** 8 10 *89* 68 45

3) Теперь начинаем справа, но начинаем с элемента, предшествующего 89, и сравниваем каждый элемент с **37** до тех пор, пока не найдем меньший чем 37, и опять поменяем местами 37 и этот элемент. Первый элемент, который меньше 37 — это 10, — меняем местами с 37. Теперь наш массив имеет вид:

12 2 6 4 *10* 8 **37** 89 68 45

4) Теперь начинаем движение с левой части массива, но начинаем с элемента, следующего за 10, и сравниваем каждый элемент с **37**, пока не обнаружим элемент больший чем **37**, после чего меняем местами **37** и этот найденный элемент. В нашем случае элементов, больших 37, не осталось, и когда мы сравним **37** с самим собой, это будет означать, что процесс закончен и элемент **37** нашел свое окончательное место.

После завершения этой операции мы имеем два неупорядоченных подмассива. Подмассив со значениями меньше 37 содержит элементы 12, 2, 6, 4, 10 и 8. Подмассив со значениями большими 37 содержит 89, 68 и 45. Сортировка продолжается путем применения алгоритма разбиения к полученным подмассивам, как это делалось с первоначальным массивом.

На основе описанного выше алгоритма напишите рекурсивную функцию **quicksort**, сортирующую одномерный целочисленный массив. Функция должна иметь параметрами целочисленный массив, начальное значение индекса и конечное значение индекса. Функция **quicksort** должна вызывать функцию **partition**, выполняющую разбиение массива.

7.25. (*Прохождение лабиринта*) Приведенная ниже сетка единиц и нулей — двумерный массив, представляющий лабиринт.

```
1 1 1 1 1 1 1 1 1 1 1 1
1 0 0 0 1 0 0 0 0 0 0 1
0 0 1 0 1 0 1 1 1 1 0 1
1 1 1 0 1 0 0 0 0 1 0 1
1 0 0 0 0 1 1 1 0 1 0 0
1 1 1 1 0 1 0 1 0 1 0 1
1 0 0 1 0 1 0 1 0 1 0 1
1 1 0 1 0 1 0 1 0 1 0 1
1 0 0 0 0 0 0 0 0 1 0 1
1 1 1 1 1 1 0 1 1 1 0 1
1 0 0 0 0 0 0 1 0 0 0 1
1 1 1 1 1 1 1 1 1 1 1 1
```

Единицами обозначены стенки, а нулями — дорожки в лабиринте.

Существует простой алгоритм прохода через лабиринт, который гарантирует, что вы найдете выход (если он, конечно, существует). Если лабиринт не имеет выхода, то вы вернетесь к тому месту, из которого вышли. Касайтесь правой рукой стены (которая находится справа от вас) и начинайте движение вперед. Все время касайтесь рукой стены. Если лабиринт поворачивает направо, вы должны следовать за поворотом стены направо. И если вы не будете отпускать руку, то в конечном счете вы доберетесь до выхода из лабиринта. Возможно, существует и более короткий путь, чем тот, которым вы прошли, но указанный метод в любом случае гарантирует вам выход из лабиринта.

Напишите рекурсивную функцию **mazeTraverse** прохода через лабиринт. Функция должна получать в качестве аргумента представляющий лабиринт массив символов 12×12 и точку входа в лабиринт. В процессе поиска выхода из лабиринта **mazeTraverse** помещает символ **X** в каждый пройденный квадрат пути вместо символа 0. Функция должна перерисовывать лабиринт после каждого перемещения, так чтобы пользователь мог наблюдать процесс решения задачи.

7.26. (*Генератор случайных лабиринтов*) Напишите функцию **mazeGenerator**, имеющую параметром двумерный символьный массив 12×12, которая генерирует при помощи случайных чисел лабиринт. Функция должна производить лабиринты, имеющие вход и выход. Испытайте вашу функцию **mazeTraverse** из упражнения 7.25 на нескольких лабиринтах.

7.27. (*Лабиринты произвольного размера*) Обобщите функции **mazeTraverse** и **mazeGenerator** из упражнений 7.25 и 7.26 для работы с лабиринтами произвольной ширины и высоты.

7.28. (*Массивы указателей на функции*) Перепишите программу, приведенную на рис. 6.22, чтобы она управлялась при помощи меню. Программа должна предлагать пользователю выбор из 4 команд примерно таким образом:

```
Выберите:
0   Вывести массив оценок
1   Найти минимальную оценку
2   Найти максимальную оценку
3   Вывести среднюю по всем тестам оценку для каждого студента
4   Выйти из программы
```

Существует одно ограничение на использование массивов указателей на функции, которое состоит в том, что все указатели должны иметь одинаковый тип. Указатели должны ссылаться на функции, возвращающие значение одного и того же типа, а также имеющие параметры одинаковых типов. По этой причине функции на рис. 6.22 должны быть изменены так, чтобы они возвращали результат одного типа и имели одинаковый набор параметров. Модифицированные функции **minimum** и **maximum** должны выводить минимальную и максимальную оценки и ничего не возвращать. Измените функцию **average** для команды меню 3, представ-

ленную на рис. 6.22, чтобы она выводила средний балл для каждого студента (а не для одного). Функция **average** не должна ничего возвращать и должна иметь список параметров, одинаковый с функциями **printArray**, **minimum** и **maximum**. Сохраните указатели на четыре функции в массиве **processGrades** и используйте выбор, сделанный пользователем, в качестве индекса массива указателей для вызова нужной функции.

7.29. (*Усовершенствования симулятора Симплетрона*) В упражнении 7.19 вы написали программную модель компьютера, которая может исполнять программы, написанные на Машинном языке Симплетрона (SML). В этом упражнении мы предлагаем вам внести несколько усовершенствований и расширений в модель Симплетрона. В упражнениях 12.26 и 12.27 мы предложим вам написать компилятор, транслирующий программы, написанные на языке высокого уровня (разновидность BASIC), в программы на Машинном языке Симплетрона. Некоторые из следующих усовершенствований и расширений могут потребоваться для выполнения программ, порожденных этим компилятором.

a) Увеличьте память Симплетрона до 1000 единиц, чтобы он мог выполнять программы большего размера.

b) Добавьте в Симплетрон возможность вычисления значений по модулю. Для этого потребуется ввести новый код операции в Машинный язык Симплетрона.

c) Добавьте в Симплетрон возможность вычисления степени числа. Для этого также потребуется ввести новый код операции в SML.

d) Замените десятичные числа шестнадцатеричными при кодировании операций в SML.

e) Добавьте в Симплетрон возможность вывода новой строки, для чего введите новый код операции в SML.

f) Внесите изменения в программу-симулятор, необходимые для того, чтобы Симплетрон, кроме целых, мог обрабатывать числа с плавающей точкой.

g) Добавьте в Симплетрон возможность ввода строк. Совет: каждое слово Симплетрона может быть разделено на две части, каждая из которых является числом из двух цифр. Используйте эти числа для представления десятичного эквивалента ASCII-кода символа. Введите в машинный язык новую операцию, которая осуществляет ввод строки и ее запись по определенному адресу в памяти Симплетрона. При этом в первую половину первого машинного слова, из числа отведенных для хранения строки, записывается количество символов в строке, т.е. длина строки. Каждая из последующих половин слов используется для хранения десятичного ASCII-кода символа. Эта новая машинная операция должна преобразовывать каждый вводимый символ в соответствующий ASCII-код и записывать его в половину слова.

h) Добавьте в Симплетрон возможность вывода строк, хранящи ' в памяти по формату, описанному в задании (g). Совет: введите в машинный язык новую инструкцию вывода строки, расположенной по

указанному адресу памяти. В первой половине первого слова записано количество символов в строке (т.е. длина строки). Каждая из последующих половин слов содержит десятичные ASCII-коды символов строки. Машинная инструкция проверяет длину строки, преобразует десятичное число в соответствующий символ и выводит его.

7.30. Что делает эта программа?

```
#include <stdio.h>

int mystery3(const char *, const char *);

main()
{
   char string1 [80], string2 [80];

   printf("Введите две строки:  ");
   scanf("%s%s", string1, string2);
   printf("Результат равен %d\n", mystery3(string1,
                                          string2));

   return 0;
}

int mystery3(const char *s1, const char *s2)
{
   for ( ; *s1 != '\0' && *s2 != '\0'; s1++, s2++)
      if (*s1 != *s2)
         return 0;

   return 1;
}
```

г л а в а



Символы и строки



Цели

- Изучить функции библиотеки для работы с символами (**ctype**).
- Научиться использовать функции из стандартной библиотеки ввода/вывода (**stdio**).
- Научиться использовать функции преобразования строк из библиотеки утилит общего назначения (**stdlib**).
- Научиться применять функции обработки строк из библиотеки для работы со строками (**string**)
- Познакомиться с мощными возможностями, предоставляемыми библиотеками функций в плане повторного использования программного кода.

Содержание

8.1. Введение
8.2. Строки и символы
8.3. Библиотека обработки символов
8.4. Функции преобразования строк
8.5. Функции стандартной библиотеки ввода/вывода
8.6. Функции операций над строками из библиотеки обработки строк
8.7. Функции сравнения из библиотеки обработки строк
8.9. Функции поиска из библиотеки обработки строк
8.10. Другие функции из библиотеки обработки строк

Резюме • Распространенные ошибки программирования • Хороший стиль программирования • Советы по переносимости программ • Упражнения для самоконтроля • Ответы к упражнениям для самоконтроля • Упражнения • Специальный раздел: более сложные упражнения по работе со строками

8.1. Введение

В этой главе мы познакомимся со стандартной библиотекой функций C, которые упрощают обработку строк и символов. Эти функции позволяют программе обрабатывать символы, строки, ввод с клавиатуры и блоки памяти. С помощью этих функций можно осуществлять операции с текстом, подобные тем, что выполняются функциями форматированного ввода/вывода **printf** и **scanf**.

Методики программирования, обсуждаемые в этой главе, используются при создании редакторов, текстовых процессоров, издательских программ макетирования, компьютерных наборных систем и других программ обработки текста.

8.2. Строки и символы

Символы — это фундаментальные «кирпичики» исходного текста программы. Любая программа составляется из последовательности символов, которые (если они сгруппированы осмысленно) воспринимаются компьютером как ряд инструкций, реализующих решение задачи. Программа может содержать *символьные константы*. Символьная константа — это величина типа **int**, представляемая в виде символа, заключенного в одинарные кавычки (апо-

строфы). Значением символьной константы является целое значение из набора символов, используемых машиной. Например, **'z'** представляет собой целое значение символа **z**, а **'\n'** представляет собой целое значение символа новой строки.

Строкой называется последовательность символов, с которой обращаются как с одним элементом. Строка может содержать буквы, цифры и различные *специальные символы*, такие как +, -, *, /, $ и другие. В C *строковые литералы*, или *строки-константы* заключаются в двойные кавычки:

```
"John Q. Doe"             (имя)
"99999 Main Street"       (дом и улица)
"Waltham, Massachusetts" (город и штат)
"(201) 555-1212"          (номер телефона)
```

Строка в C является массивом символов, который заканчивается *нулевым символом* (**'\0'**). Доступ к строке осуществляется через указатель, ссылающийся на первый символ строки. Значением строки является адрес ее первого символа. Таким образом, в C правомерно сказать, что *строка — это указатель*, фактически указатель на первый символ строки. В этом смысле строка аналогична массиву, поскольку массив также является указателем на его первый элемент.

Строка может быть присвоена в объявлении либо массиву символов, либо переменной типа **char ***. Каждое из объявлений

```
char color [] = "blue";
char *colorPtr = "blue";
```

инициализирует переменную строкой **"blue"**. Первое объявление создает массив **color**, состоящий из 5 элементов и содержащий символы: **'b'**, **'l'**, **'u'**, **'e'**, **'\0'**. Второе объявление создает переменную-указатель **colorPtr**, который указывает на строку **"blue"**, находящуюся в каком-то месте памяти.

Совет по переносимости программ 8.1

Если переменная типа **char*** инициализируется строковым литералом, то некоторые компиляторы могут поместить такую строку в неизменяемую область памяти. Если, например, вам требуется изменять какие-то символы в литеральной строке, то, чтобы иметь такую возможность на всех системах, строку нужно запомнить в массиве символов.

Приведенное выше объявление массива можно также записать следующим образом:

```
char color [] = {'b', 'l', 'u', 'e', '\0'};
```

При объявлении массива символов для хранения строки он должен иметь достаточный размер, чтобы вместить строку и ограничивающий нулевой символ (**NULL**). В приведенном объявлении размер массива определяется автоматически, исходя из числа инициализаторов в списке.

Распространенная ошибка программирования 8.1

При определении размера массива символов не выделено место, чтобы запомнить символ **NULL**, который ограничивает строку.

Распространенная ошибка программирования 8.2

Попытка напечатать «строку», которая не содержит ограничивающего символа **NULL**.

Хороший стиль программирования 8.1

Когда вы объявляете массив символов для хранения строки, убедитесь, что размер его достаточен для хранения самой большой строки, с которой предполагается работать. Язык С позволяет запоминать строку любой длины. Если строка оказалась длиннее массива, в котором она должна сохраняться, то символы, выходящие за размер массива, перепишут данные в области памяти, следующей сразу за массивом.

Массиву можно присвоить строку, используя функцию **scanf**. Например, следующий оператор присваивает строку массиву символов **word[20]**.

```
scanf("%s", word);
```

Строка, введенная пользователем, запоминается в **word** (отметьте, что **word** является массивом, который, разумеется, есть указатель и, следовательно, нет необходимости применять & к аргументу **word**). Функция **scanf** будет читать символы до тех пор, пока не встретит символ пробела, новой строки или признак конца файла. Обратите внимание, что длина строки не должна превышать 19 символов, чтобы оставалось место для ограничивающего строку символа **NULL**. Чтобы можно было напечатать массив символов как строку, в конце массива обязательно должен содержаться символ **NULL**.

Распространенная ошибка программирования 8.3

Обработка одиночного символа в качестве строки. Строка — это указатель, который, возможно, представляется данными типа **long int**. Символ является небольшим целым числом (значения кодовой таблицы ASCII находятся в диапазоне 0-255). На многих системах такая замена вызовет ошибку, поскольку младшие адреса памяти резервируются для специального использования — например, для адресации программ обработки прерываний операционной системы, — так что произойдет «несанкционированный доступ».

Распространенная ошибка программирования 8.4

Передача символа в качестве аргумента функции, которая оперирует со строками.

Распространенная ошибка программирования 8.5

Передача строки в качестве аргумента функции, которая оперирует с символами.

8.3. Библиотека обработки символов

Библиотека обработки символов включает в себя несколько функций, выполняющих ряд полезных проверок и операций с символьными данными. Каждая функция получает в качестве аргумента символ — представляемый типом **int** — или индикатор **EOF** (конец файла). Как мы уже говорили в 4-й главе, операции с символами часто выполняются как с целыми числами, поскольку символ в С это однобайтовое целое. Вспомните, что **EOF** обычно имеет значение -1, а архитектура некоторых аппаратных средств не позволяет запоминать отрицательные значения в переменных типа **char**. Следовательно, функции обработки символов оперируют с символами как с целыми числами. На рис. 8.1 приведен перечень функций библиотеки обработки символов.

Прототип	Описание функции
`int isdigit (int c)`	Возвращает значение true, если **c** является цифрой, и 0 (false) в других случаях.
`int isalpha (int c)`	Возвращает значение true, если **c** является буквой, и 0 в других случаях.
`int isalnum (int c)`	Возвращает значение true, если **c** является цифрой или буквой, и 0 в других случаях.
`int isxdigit (int c)`	Возвращает значение true, если **c** является одним из символов шестнадцатеричного формата, и 0 в других случаях. Смотрите приложение Д, «Системы счисления», где детально обсуждаются восьмеричная, десятичная и шестнадцатеричная системы счисления.
`int islower (int c)`	Возвращает значение true, если **c** является буквой нижнего регистра, и 0 в других случаях.
`int isupper (int c)`	Возвращает значение true, если **c** является буквой верхнего регистра, и 0 в других случаях.
`int tolower (int c)`	Если **c** является буквой верхнего регистра, то **tolower** возвращает **c** как букву нижнего регистра. В других случаях **tolower** возвращает аргумент без изменений.
`int toupper (int c)`	Если **c** является буквой нижнего регистра, то **toupper** возвращает **c** как букву верхнего регистра. В других случаях **toupper** возвращает аргумент без изменений.
`int isspace (int c)`	Возвращает значение true, если **c** является пробельным символом – пробел (' '), новая страница ('\f'), новая строка ('\n'), возврат каретки ('\r'), горизонтальная табуляция ('\t') или вертикальная табуляция ('\v'), – и 0 в других случаях.
`int iscntrl (int c)`	Возвращает значение true, если **c** является управляющим символом, и 0 в других случаях.
`int ispunct (int c)`	Возвращает значение true, если **c** является отображаемым при печати символом, но не относится к пробельным символам, цифрам или буквам. В других случаях функция возвращает значение 0.
`int isprint (int c)`	Возвращает значение true, если **c** является отображаемым при печати символом, включая символ пробела (' '), и 0 в других случаях.
`int isgraph (int c)`	Возвращает значение true, если **c** является отображаемым при печати символом, исключая символ пробела (' '), и 0 в других случаях.

Рис. 8.1. Перечень функций библиотеки обработки символов

Хороший стиль программирования 8.2

Если вы вызываете функции библиотеки обработки символов, подключите заголовочный файл **<ctype.h>**.

Программа, приведенная на рис. 8.2, демонстрирует применение функций ***isdigit***, ***isalpha***, ***isalnum*** и ***isxdigit***. Функция ***isdigit*** определяет, является ли ее аргумент цифрой **(0-9)**. Функция ***isalpha*** определяет, попадает ли ее аргумент в список букв верхнего **(A-Z)** или нижнего **(a-z)** регистров. Функция ***isalnum*** определяет принадлежность аргумента буквам верхнего и нижнего регистров или цифрам. Функция ***isxdigit*** определяет, является ли аргумент шестнадцатеричной цифрой **(A-F, a-f, 0-9)**.

```
/* Применение функций isdigit, isalpha, isalnum и isxdigit */
#include <stdio.h>
#include <ctype.h>

main()
{
   printf("%s\n%s%s\n%s%s\n\n", "According to isdigit: ",
      isdigit('8') ? "8 is a " : "8 is not a ", "digit",
      isdigit('#') ? "# is a " : "# is not a ", "digit");
   printf("%s\n%s%s\n%s%s\n%s%s\n%s%s\n\n",
          "According to isalpha:",
      isalpha('A') ? "A is a " : "A is not a ", "letter",
      isalpha('b') ? "b is a " : "b is not a ", "letter",
      isalpha('&') ? "& is a " : "& is not a ", "letter",
      isalpha('4') ? "4 is a " : "4 is not a ", "letter");
   printf("%s\n%s%s\n%s%s\n%s%s\n\n", "According to isalnum:",
      isalnum('A') ? "A is a " : "A is not a ",
                     "digit or a letter",
      isalnum('8') ? "8 is a " : "8 is not a ",
                     "digit or a letter",
      isalnum('#') ? "# is a " : "# is not a ",
                     "digit or a letter");
   printf("%s\n%s%s\n%s%s\n%s%s\n%s%s\n%s%s\n",
           "According to isxdigit:",
      isxdigit('F') ? "F is a " : "F is not a ",
                      "hexadecimal digit",
      isxdigit('J') ? "J is a " : "J is not a ",
                      "hexadecimal digit",
      isxdigit('7') ? "7 is a " : "7 is not a ",
                      "hexadecimal digit",
      isxdigit('$') ? "$ is a " : "$ is not a ",
                      "hexadecimal digit",
      isxdigit('f') ? "f is a " : "f is not a ",
                      "hexadecimal digit");
   return 0;
}
```

```
According to isdigit:
8 is a digit
# is not a digit

According to isalpha:
A is a letter
b is a letter
& is not a letter
4 is not a letter

According to isalnum:
A is a digit or a letter
8 is a digit or a letter
# is not a digit or a letter

According to isxdigit:
F is a hexadecimal digit
J is not a hexadecimal digit
7 is a hexadecimal digit
$ is not a hexadecimal digit
f is a hexadecimal digit
```

Рис. 8.2. Применение **isdigit**, **isalpha**, **isalnum**, **isxdigit**

Программа рис. 8.2 использует с каждой функцией условную операцию (?:), чтобы определить, какую из строк **« is a » (это)** или **« is not a » (это не)** следует вывести на печать для каждого проверяемого символа. Например, выражение

```
isdigit ('8') ? "8 is a" : "8 is not a"
```

означает, что если '**8**' является цифрой, т.е. функция **isdigit** возвращает истинное значение (ненулевое), то печатается строка **« 8 is a »**, и, если '**8**' не является цифрой, т.е. **isdigit** возвращает **0**, печатается строка **« 8 is not a »**.

Программа, приведенная на рис. 8.3, демонстрирует применение функций ***islower***, ***isupper***, ***tolower*** и ***toupper***. Функция ***islower*** определяет, является ли ее аргумент буквой нижнего регистра (**a–z**). Функция ***isupper*** определяет, является ли ее аргумент буквой верхнего регистра (**A–Z**). Функция ***tolower*** преобразует регистр буквы из верхнего в нижний и возвращает букву нижнего регистра. Если аргумент не является буквой верхнего регистра, то функция ***tolower*** возвращает аргумент без изменений. Функция ***toupper*** преобразует регистр буквы из нижнего в верхний и возвращает букву верхнего регистра. Если аргумент не является буквой нижнего регистра, то функция ***toupper*** возвращает аргумент без изменений.

```
/* Применение функций islower, isupper, tolower, toupper */
#include <stdio.h>
#include <ctype.h>

main()
{
   printf("%s\n%s%s\n%s%s\n%s%s\n%s%s\n\n",
          "According to islower:",
          islower ('p') ? "p is a " : "p is not a ",
          "lowercase letter",
          islower ('p') ? "p is a " : "p is not a ",
          "lowercase letter",
          islower ('5') ? "5 is a " : "5 is not a ",
          "lowercase letter",
          islower ('!') ? "! is a " : "! is not a ",
          "lowercase letter" ) ;
   printf("%s\n%s%s\n%s%s\n%s%s\n%s%s\n\n",
          "According to isupper:",
          isupper ('d') ? "D is a " : "D is not an ",
          "uppercase letter",
          isupper ('d') ? "d is a " : "d is not an ",
          "uppercase letter",
          isupper ('8') ? "8 is a " : "8 is not an ",
          "uppercase letter",
          isupper ('$') ? "$ is a " : "$ is not an ",
          "uppercase letter" );
   printf("%s%c\n%s%c\n%s%c\n%s%c\n",
          "u converted to uppercase is ", toupper ('u'),
          "7 converted to uppercase is ", toupper ('7'),
          "$ converted to uppercase is ", toupper ('$'),
          "L converted to lowercase is ", tolower ('L'));
   return 0;
}
```

Рис. 8.3. Применение **islower**, **isupper**, **tolower**, **toupper** (часть 1 из 2)

```
According to islower:
p is a lowercase letter
P is not a lowercase letter
5 is not a lowercase letter
! is not a lowercase letter

According to isupper:
D is an uppercase letter
d is not an uppercase letter
8 is not an uppercase letter
$ is not an uppercase letter

u converted to uppercase is U
7 converted to uppercase is 7
$ converted to uppercase is $
L converted to lowercase is l
```

Рис. 8.3. Применение **islower**, **isupper**, **tolower**, **toupper** (часть 2 из 2)

Рис. 8.4 иллюстрирует применение функций ***isspace***, ***iscntrl***, ***ispunct***, ***isprint*** и ***isgraph***. Функция ***isspace*** определяет принадлежность аргумента следующему списку пробельных символов: пробел (' '), новая страница ('\f'), новая строка ('\n'), возврат каретки ('\r'), горизонтальная табуляция ('\t'), вертикальная табуляция ('\v'). Функция ***iscntrl*** определяет принадлежность аргумента следующему списку управляющих символов: горизонтальная табуляция ('\t'), вертикальная табуляция ('\v'), новая страница ('\f'), звуковой сигнал ('\a'), возврат на один символ ('\b'), возврат каретки ('\r'), новая строка ('\n'). Функция ***ispunct*** определяет, является ли аргумент знаком пунктуации или специальным символом, не относящимся к цифрам, буквам или пробелам, например, $, #, (,), [,], {, }, ;, :, % и т.д. Функция ***isprint*** определяет, можно ли отобразить данный символ на экране (включая символ пробела). Функция ***isgraph*** работает так же, как и ***isprint***, но из списка символов исключен символ пробела.

```
/* Применение функций isspace, iscntrl, ispunct, isprint,
   isgraph */
#include <stdio.h>
#include <ctype.h>

main()
{
   printf("%s\n%s%%s%s\n%s%s%s\n%s%s\n\n",
          "According to isspace:",
          "Newline", isspace('\n') ? " is a " : " is not a ",
          "whitespace character" , "Horizontal tab" ,
          isspace ('\t') ? " is a " : " is not a ",
          "whitespace character",
          isspace ('%') ? "% is a " : "% is not a ",
          "whitespace character");
   printf("%s\n%s%s%s\n%s%s\n\n", "According to iscntrl:",
          "Newline", iscntrl ('\n') ? " is a " : " is not a ",
          "control character",
          iscntrl ('$') ? "$ is a " : "$ is not a ",
          "control character");
```

Рис. 8.4. Применение **isspace**, **iscntrl**, **ispunct**, **isprint**, **isgraph** (часть 1 из 2)

```
   printf("%s\n%s%s\n%s%s\n%s%s\n\n", "According to ispunct:",
          ispunct (';') ? "; is a " : "; is not a ",
          "punctuation character",
          ispunct ('Y') ? "Y is a " : "Y is not a ",
          "punctuation character",
          ispunct ('#') ? "# is a " : "# is not a ",
          "punctuation character");
   printf("%s\n%s%s\n%s%s%s\n\n", "According to isprint:",
          isprint ('$') ? "$ is a " : "$ is not a ",
          "printing character"
          "Alert", isprint ('\a') ? " is a " : " is not a ",
          "printing character");
   printf("%s\n%s%s\n%s%s%s\n", "According to isgraph:",
          isgraph ('Q') ? "Q is a " : "Q is not a ",
          "printing character other than a space",
          "Space", isgraph(' ') ? " is a " : " is not a ",
          "printing character other than a space");
   return 0;
}
```

```
According to isspace:
Newline is a whitespace character
Horizontal tab is a whitespace character
% is not a whitespace character

According to iscntrl:
Newline is a control character
$ is not a control character

According to ispunct:
; is a punctuation character
Y is not a punctuation character
# is a punctuation character

According to isprint:
$ is a printing character
Alert is not a printing character

According to isgraph:
Q is a printing character other than a space
Space is not a printing character other than a space
```

Рис. 8.4. Применение **isspace**, **iscntrl**, **ispunct**, **isprint**, **isgraph** (часть 2 из 2)

8.4. Функции преобразования строк

В этом разделе рассматриваются *функции преобразования строк* из библиотеки утилит общего назначения (***stdlib***). Эти функции преобразуют строки цифр в целые значения и значения с плавающей точкой. На рис. 8.5 приведен перечень функций преобразования строк. Обратите внимание на модификатор **const** в описании переменной **nPtr** в заголовке функции (читается справа налево: «**nPtr** является указателем на символьную константу»); **const** объявляет, что значение аргумента не будет изменяться.

Прототип	Описание функции
`double atof (const char *nPtr)`	Преобразует строку **nPtr** в тип **double**.
`int atoi (const char *nPtr)`	Преобразует строку **nPtr** в тип **int**.
`long atol (const char *nPtr)`	Преобразует строку **nPtr** в тип long **int**.
`double strtod (const char *nPtr, char **endPtr)`	Преобразует строку **nPtr** в тип **double**.
`long strtol (const char *nPtr, char **endPtr, int base)`	Преобразует строку **nPtr** в тип **long**.
`unsigned long strtoul (const char *nPtr, char **endPtr, int base)`	Преобразует строку **nPtr** в тип **unsigned long**.

Рис. 8.5. Перечень функций преобразования строк из библиотеки утилит общего назначения

Хороший стиль программирования 8.3

Когда вы вызываете функции из библиотеки утилит общего назначения, подключите заголовочный файл **<stlib.h>**.

Функция ***atof*** (рис. 8.6) преобразует аргумент — строку, которая представляет число с плавающей точкой, — в значение типа **double** и возвращает значение типа **double**. Если аргумент функции не может быть преобразован — например, если первый символ строки не является цифрой, — поведение функции **atof** не определено.

```
/* Применение atof */
#include <stdio.h>
#include <stdlib.h>

main()
{
   double d;

   d = atof("99.0") ;
   printf("%s%.3f\n%s%.3f\n",
          "The string \"99.0\" converted to double is ", d,
          "The converted value divided by 2 is ", d / 2.0);
   return 0;
}
```

```
The string "99.0" converted to double is 99.000
The converted value divided by 2 is 49.500
```

Рис. 8.6. Применение **atof**

Функция ***atoi*** (рис. 8.7) преобразует аргумент — строку цифр, которая представляет целое число, — в значение типа **int** и возвращает значение типа **int**. Если аргумент функции не может быть преобразован, то поведение функции **atoi** не определено.

```
/* Применение atoi */
#include <stdio.h>
#include <stdlib.h>

main()
{
   int I;

   i = atoi("2593")
   printf("%s%d\n%s%d\n",
          "The string \"2593\" converted to int is ", I,
          "The converted value minus 593 is ", i - 593);
   return 0;
}
```

```
The string "2593" converted to int is 2593
The converted value minus 593 is 2000
```

Рис. 8.7. Применение **atoi**

Функция ***atol*** (рис. 8.8) преобразует аргумент — строку цифр, которая представляет число типа длинное целое, — в значение типа **long** и возвращает значение типа **long**. Если аргумент функции не может быть преобразован, то поведение функции **atol** не определено. Если для хранения типа **int** и типа **long** используются 4 байта, то функции **atoi** и **atol** работают идентично.

```
/* Применение atol */
#include <stdio.h>
#include <stdlib.h>

main()
{
   long l;

   l = atol("1000000");
   printf("%s%ld\n%s%ld\n",
          "The string \"1000000\" converted to long int is ", l,
          "The converted value divided by 2 is ", l / 2);
   return 0;
}
```

```
The string "1000000" converted to long int is 1000000
The converted value divided by 2 is 500000
```

Рис. 8.8. Применение **atol**

Функция ***strtod*** (рис. 8.9) преобразует последовательность символов, представляющих число с плавающей точкой, в значение типа **double**. Эта функция имеет два аргумента: строку **(char *)** и указатель строки. Строка содержит последовательность символов, которые должны быть преобразованы в тип **double**. Указателю присваивается адрес символа, который является первым символом строки-остатка после преобразования части строки. Оператор

```
d = strtod (string, &stringPtr);
```

из программы рис. 8.9 означает, что **d** присваивается значение типа **double**, полученное после преобразования строки **string**, а **stringPtr** будет указывать на первый символ (%) после преобразования значения (**51.2**) из **string**.

```
/* Применение strtod */
#include <stdio.h>
#include <stdlib.h>

main()
{
   double d;
   char *string = "51.2% are admitted";
   char *stringPtr;

   d = strtod(string, &stringPtr);
   printf("The string \"%s\" is converted to the \n",
          string) ;
   printf("double value %.2f and the string \"%s\"\n",
          d, stringPtr);
   return 0;
}
```

```
The string "51.2% are admitted" is converted to the
double value 51.20 and the string "% are admitted"
```

Рис. 8.9. Применение **strtod**

Функция ***strtol*** (рис. 8.10) преобразует последовательность символов, представляющих целое число, в значение типа **long**. Эта функция имеет три аргумента: строку (**char ***), указатель строки и целое число. Строка содержит последовательность символов, которые должны быть преобразованы. Указателю присваивается адрес символа, который является первым символом строки-остатка после преобразования части строки. Целое число определяет *основание*, по которому производится преобразование. Оператор

```
x = strtol (string, &remainderPtr, 0);
```

из программы рис. 8.10 означает, что **x** присваивается значение типа **long**, полученное после преобразования строки **string**. Второй аргумент — **remainderPtr** — после преобразования будет указывать на остаток строки. Если для второго аргумента использовать значение **NULL**, то строка-остаток будет проигнорирована. Третий аргумент — **0** — показывает, что преобразуемое значение может иметь восьмеричный (основание 8), десятичный (основание 10) или шестнадцатеричный формат (основание 16). Основание может иметь значение 0 или может быть любым значением в диапазоне от 2 до 36. См. приложение Д, «Системы счисления», где детально обсуждаются восьмеричная, десятичная и шестнадцатеричная системы счисления. Для численного представления целых значений от 11 до 35 в диапазоне оснований от 11 до 36 используются символы A–Z. Например, шестнадцатеричное число может содержать цифры 0–9 и символы A–F. Целые по основанию 11 могут содержать цифры 0–9 и символ A. Целые по основанию 24 могут содержать цифры 0–9 и символы A–N. Целые по основанию 36 могут содержать цифры 0–9 и символы A–Z.

```
/* Применение strtol */
#include <stdio.h>
#include <stdlib.h>

main()
{
   long x;
   char *string = "-1234567abc", *remainderPtr;

   x = strtol(string, &remainderPtr, 0);
   printf("%s\"%s\"\n%s%ld\n%s\"%s\"\n%s%ld\n",
          "The original string is ", string,
          "The converted value is ", x,
          "The remainder of the original string is ",
          remainderPtr,
          "The converted value plus 567 is ", x + 567);
   return 0;
}
```

```
The original string is "-1234567abc"
The converted value is -1234567
The remainder of the original string is "abc"
The converted value plus 567 is -1234000
```

Рис. 8.10. Применение **strtol**

Функция ***strtoul*** преобразует последовательность символов, представляющую целое типа **unsigned long**, в значение типа **unsigned long**. Эта функция работает аналогично функции **strtol**. Оператор

```
x = strtoul (string, &remainderPtr, 0);
```

из программы рис. 8.11 означает, что x присваивается значение **unsigned long**, полученное после преобразования **string**. Второй аргумент, **&remainderPtr**, после преобразования указывает на остаток **string**. Третий аргумент — **0** — показывает, что преобразуемое значение может иметь восьмеричный, десятичный или шестнадцатеричный формат.

```
/* Применение strtoul */
#include <stdio.h>
#include <stdlib.h>

main()
{
   unsigned long x;
   char *string = "1234567abc", *remainderPtr;
   x = strtoul(string, &remainderPtr, 0);
   printf("%s\"%s\"\n%s%lu\n%s\"%s\"\n%s%lu\n",
          "The original string is ", string,
          "The converted value is ", x,
          "The remainder of the original string is ",
          remainderPtr,
          "The converted value minus 567 is ", x - 567);
   return 0;
}
```

Рис. 8.11. Применение **strtoul** (часть 1 из 2)



```
The original string is "1234567abc"
The converted value is 1234567
The remainder of the original string is "abc"
The converted value minus 567 is 1234000
```

Рис. 8.11. Применение **strtoul** (часть 2 из 2)

8.5. Функции стандартной библиотеки ввода/вывода

В этом разделе представлены несколько функций из стандартной библиотеки ввода/вывода **(stdio)**, предназначенные специально для операций с символьными и строковыми данными. На рис. 8.12 приведен перечень функций ввода/вывода символов и строк из стандартной библиотеки ввода/вывода.

Прототип	Описание функции
`int getchar (void)`	Вводит следующий символ со стандартного устройства ввода и возвращает его в формате целого.
`char *gets (char *s)`	Вводит символы со стандартного устройства ввода в массив **s** до тех пор, пока не встретит символ новой строки или индикатор конца файла. После этого к массиву добавляется ограничивающий символ **NULL**.
`int putchar (int c)`	Печать символа, хранящегося в **c**.
`int puts (const char *s)`	Печать строки **s** с последующим символом новая строка.
`int sprintf (char *s, const char *format, ...)`	Эквивалент функции **printf** за исключением того, что результат вывода запоминается в массиве **s**, а не отображается на экране.
`int sscanf (char *s, const char *format, ...)`	Эквивалент функции **scanf** за исключением того, что ввод осуществляется из массива **s**, а не с клавиатуры.

Рис. 8.12. Символьные и строковые функции стандартной библиотеки ввода/вывода

Хороший стиль программирования 8.4

Если вы вызываете функции из стандартной библиотеки ввода/вывода, подключите заголовочный файл **<stdio.h>**.

Программа, приведенная на рис. 8.13, использует функции **gets** и **putchar** для чтения строки текста со стандартного входного устройства (клавиатуры) и рекурсивно выводит символы строки в обратном порядке. Функция **gets** читает символы со стандартного входного устройства и передает их своему аргументу — массиву типа **char** — до тех пор, пока не встретит символ новой строки или индикатор конца файла. Символ **NULL** (**'\0'**) добавляется в массив после окончания считывания. Функция **putchar** печатает свой аргумент — символ. Для печати строки текста в обратном порядке программа рис. 8.13 вызывает рекурсивную функцию **reverse**. Если первый символ массива, полученный функцией **reverse**, является нулевым символом **'\0'**, то следует возврат из функции. В противном случае следует вызов **reverse**, и ее аргумент указывает на адрес «нового» массива, который начинается с элемента **s[1]**. Когда рекурсивный вызов завершается, функция **putchar** выводит элемент **s[0]**. Порядок

следования двух операторов в части **else** структуры **if** приводит к тому, что функция reverse переходит к граничному символу **NULL** строки до вывода символа на печать. Как только рекурсивный вызов завершается, символы выводятся в обратном порядке.

```
/* Применение gets and putchar */
#include <stdio.h>

main()
{
   char sentence[80];
   void reverse(char *);

   printf("Enter a line of text:\n");
   gets(sentence);

   printf("\nThe line printed backwards is:\n");
   reverse(sentence);

   return 0;
}

void reverse(char *s)
{
   if (s[0] == '\0')
      return;
   else {
      reverse(&s[1]);
      putchar(s[0]);
   }
}
```

```
Enter a line of text:
Characters and Strings

The line printed backwards is:
sgnirtS dna sretcarahC

Enter a line of text:
able was I ere I saw elba

The line printed backwards is:
able was I ere I saw elba
```

Рис. 8.13. Применение **gets** and **putchar**

Программа, приведенная на рис. 8.14, использует функции **getchar** и **puts** для чтения символов со стандартного входного устройства в символьный массив **sentence** и выводит массив символов как строку. Функция **getchar** читает символ со стандартного входного устройства и возвращает его как целое значение. Функция **puts** получает в качестве аргумента строку **(char *)** и выводит ее с последующим символом новой строки (**'\n'**).

```
/* Применение getchar и puts */
#include <stdio.h>

main()
{
   char c, sentence[80];
   int i = 0;

   puts("Enter line of text:");
   while ( ( c = getchar() ) != '\n')
      sentence[i++] = c;

   sentence[i] = '\0';  /* вставьте символ NULL в конец строки */
   puts("\nThe line entered was:");
   puts(sentence);
   return 0;
}
```

```
Enter line of text:
This is a test.

The line entered was:
This is a test.
```

Рис. 8.14. Применение **getchar** and **puts**

Программа прекращает ввод символов, когда **getchar** считывает введенный пользователем символ новой строки. В массив **sentence** добавляется символ **NULL**, чтобы массив мог восприниматься как строка. Функция **puts** выводит строку, содержащуюся в **sentence**.

Программа, приведенная на рис. 8.15, использует функцию **sprintf** для печати форматированных данных в массив **s** — массив символов. Эта функция использует ту же самую спецификацию преобразования, что и **printf** (см. главу 9, где детально обсуждаются все возможности форматированной печати). Программа вводит значения типа **int** и **float**, которые форматируются и выводятся в массив **s**. Массив **s** является первым аргументом функции **sprintf**.

```
/* Применение sprintf */
#include <stdio.h>

main ()
{
   char s[80];
   int x;
   float y;

   printf("Enter an integer and float:\n");
   scanf("%d%f", &x, &y);
   sprintf(s, "Integer:%6d\nFloat:%8.2f", x, y);
   printf("%s\n%s\n",
          "The formatted output stored in array s:", s);
   return 0;
}
```

Рис. 8.15. Применение **sprintf** (часть 1 из 2)

```
Enter an integer and float:
298 87.375
The formatted output stored in array s:
Integer:   298
Float:   87.38
```

Рис. 8.15. Применение **sprintf** (часть 2 из 2)

Программа, приведенная на рис. 8.16, демонстрирует функцию **sscanf** для чтения форматированных данных из массива символов **s**. Эта функция использует ту же самую спецификацию преобразования, что и **scanf**. Программа считывает значения типа **int** и **float** из массива **s** и запоминает величины соответственно в переменных **x** и **y**. Величины **x** и **y** выводятся на печать. Массив **s** является первым аргументом функции **sscanf**.

```
/* Применение sscanf */
#include <stdio.h>

main()
{
   char s[] = "31298 87.375";
   int x;
   float y;

   sscanf(s, "%d%f", &x, &y);
   printf("%s\n%s%6d\n%s%8.3f\n",
          "The values stored in character array s are:",
          "Integer:", x, "Float:", y);
   return 0;
}
```

```
The values stored in character array s are:
Integer: 31298
Float:  87.375
```

Рис. 8.16. Применение **sscanf**

8.6. Функции операций над строками из библиотеки обработки строк

Библиотека обработки строк содержит множество полезных функций для выполнения операций со строковыми данными, для сравнения строк, для поиска символов и других строк в строке, для деления строки на лексемы (логические части) и определения длины строки. В этом разделе представлены функции для выполнения операций над строками из библиотеки обработки строк. Перечень этих функций приведен на рис. 8.17.

Хороший стиль программирования 8.5

Если вы вызываете функции из библиотеки обработки строк, подключите заголовочный файл **<string.h>**.

Прототип	Описание функции
`char *strcpy (char *s1, const char *s2)`	
	Копирует строку **s2** в массив **s1**. Возвращает значение **s1**.
`char *strncpy (char *s1, const char *s2, size_t n)`	
	Копирует не более чем **n** символов строки **s2** в массив **s1**. Возвращает значение **s1**.
`char *strcat (char *s1, const char *s2)`	
	Объединяет строку **s2** со строкой массива **s1**. Первый символ строки **s2** переписывает символ **NULL** строки **s1**. Возвращает значение **s1**.
`char *strncat (char *s1, const char *s2, size_t n)`	
	Объединяет не более чем **n** символов строки **s2** со строкой массива **s1**. Первый символ строки **s2** переписывает символ **NULL** строки **s1**. Возвращает значение **s1**.

Рис. 8.17. Функции для выполнения операций над строками из библиотеки обработки строк

Функция ***strcpy*** копирует второй аргумент — строку — в первый аргумент — массив символов, который должен иметь достаточный размер, чтобы запомнить строку и ограничивающий символ **NULL**, который также копируется. Функция ***strncpy*** эквивалентна **strcpy** за исключением того, что **strncpy** задает число символов, которые необходимо скопировать из строки в массив. Обратите внимание, что функция **strncpy** не копирует ограничивающий символ **NULL** второго аргумента, если не выполнено следующее условие: число копируемых символов превышает длину строки, по меньшей мере, на единицу. Например, если вторым аргументом является строка **"test"**, то ограничивающий символ **NULL** записывается, только если значение третьего аргумента *функции составляет не менее 5 (4 символа в строке* **"test"** *плюс 1 символ* **NULL**). Если значение третьего аргумента больше **5**, то символы **NULL** добавляются в массив до тех пор, пока общее число записанных символов не станет равным числу, заданному в третьем аргументе.

Распространенная ошибка программирования 8.6

Отсутствует ограничивающий символ **NULL** *в первом аргументе функции* ***strncpy***, *когда величина третьего аргумента меньше или равна длине строки во втором аргументе.*

Программа, приведенная на рис. 8.18, вызывает функцию **strcpy** для копирования всей строки из массива **x** в массив **y** и первых 14 символов из массива **x** в массив **z**. Символ **NULL** (**'\0'**) добавляется в массив **z** в отдельном операторе, поскольку обращение программы к функции **strcpy** не копирует нулевой символ (третий аргумент меньше длины строки второго аргумента).

Функция ***strcat*** выполняет объединение строк (*конкатенацию*): второго аргумента — строки — с первым аргументом — массивом символов, содержащим строку. Первый символ второго аргумента заменяет последний символ **NULL** строки первого аргумента. Программист должен убедиться, что массив, содержащий первую строку, достаточен по размерам для хранения обеих строк и ограничивающего символа **NULL**, который копируется из второй стро-

ки. Функция ***strncat*** выполняет объединение определенного числа символов из второй строки с первой строкой. Ограничивающий символ **NULL** добавляется к строке результата автоматически. Программа, приведенная на рис. 8.19, демонстрирует применение функций ***strcat*** и ***strncat***.

```
/* Применение strcpy и strncpy */
#include <stdio.h>
#include <string.h>

main ()
{
   char x[] = "Happy Birthday to You";
   char y[25], z[15];

   printf ("%s%s\n%s%s\n",
           "The string in array x is: ", x,
           "The string in array y is: ", strcpy(y, x));

   strncpy(z, x, 14);
   z[14] = '\0';
   printf ("The string in array z is: %s\n", z);
   return 0;
}
```

```
The string in array x: Happy Birthday to You
The string in array y: Happy Birthday to You
The string in array z: Happy Birthday
```

Рис. 8.18. Применение **strcpy** и **strncpy**

```
/* Применение strcat и strncat */
#include <stdio.h>
#include <string.h>

main()
{
   char s1[20] = "Happy ";
   char s2[] = "New Year";
   char s3[40] = "";

   printf("s1 = %s\ns2 = %s\n", s1, s2);
   printf("strcat(s1, s2) = %s\n", strcat(s1, s2));
   printf("strncat(s3, s1, 6) = %s\n", strncat(s3, s1, 6));
   printf ("strcat(s3, s1) = %s\n", strcat(s3, s1));
   return 0;
}
```

```
s1 = Happy
s2 = New Year
strncat(s1, s2) = Happy New Year
strncat(s3, s1, 6) = Happy
strcat(s3, s1) = Happy Happy New Year
```

Рис. 8.19. Применение **strcat** и **strncat**

8.7. Функции сравнения из библиотеки обработки строк

В этом разделе представлены функции сравнения строк **strcmp** и **strncmp** из библиотеки обработки строк. Прототипы функций и краткое описание каждой из них приведены на рис. 8.20.

Прототип	Описание функции
`int strcmp (const char *s1, const char *s2)`	
	Сравнивает строку **s1** со строкой **s2**. Функция возвращает 0, значение меньше 0 или больше 0, если **s1** соответственно равна, меньше или больше **s2**.
`int strncmp (const char *s1, const char *s2, size_t n)`	
	Сравнивает до **n** символов строки s1 со строкой **s2**. Функция возвращает 0, значение меньше 0 или больше 0, если **s1** соответственно равна, меньше или больше чем **s2**.

Рис. 8.20. Функции сравнения строк из библиотеки обработки строк

Программа, показанная на рис. 8.21, сравнивает три строки, используя функции ***strcmp*** и ***strncmp***. Функция **strcmp** сравнивает символ за символом первый аргумент-строку со вторым аргументом-строкой. Функция возвращает 0, если строки равны, отрицательное значение, если первая строка меньше второй, и положительное значение, если первая строка больше второй. Функция **strncmp** эквивалентна функции **strcmp**, за исключением того, что **strncmp** сравнивает до **n** символов в строках. Функция **strncmp** не сравнивает символы, идущие за символом **NULL**. Эта программа печатает целое значение, возвращаемое при каждом вызове функции.

Распространенная ошибка программирования 8.7

Предположение, что **strcmp** и **strncmp** возвращают 1 в случае равенства строк. На самом деле функции **strcmp** и **strncmp** возвращают в этом случае 0 (значение false в C) и, следовательно, при проверке двух строк на равенство результат должен сравниваться с 0.

Чтобы понять утверждение, что одна строка «больше» или «меньше» другой, рассмотрим процесс алфавитной сортировки ряда фамилий. Читатель, несомненно, поместил бы фамилию «Jones» перед фамилией «Smith», поскольку первая буква фамилии «Jones» стоит в алфавите перед первой буквой фамилии «Smith». Но алфавит — это не просто набор из 26 букв; это упорядоченный список символов. Каждая буква занимает свое определенное место в списке. «Z» — не просто буква алфавита, а двадцать шестая буква алфавита.

Как компьютер узнает, что одна буква идет перед другой? Все символы представлены в компьютере в виде числовых кодов; когда компьютер сравнивает две строки, он фактически сравнивает числовые коды символов этих строк.

```
/* Применение strcmp и strncmp */
#include <stdio.h>
#include <string.h>

main()
{
   char *s1 = "Happy New Year";
   char *s2 = "Happy New Year";
   char *s3 = "Happy Holidays";

   printf("%s%s\n%s%s\n%s%s\n\n%s%2d\n%s%2d\n%s%2d\n\n",
          "s1 = ", s1, "s2 = ", s2, "s3 = ", s3,
          "strcmp(s1, s2) = ", strcmp(s1, s2),
          "strcmp(s1, s3) = ", strcmp(s1, s3),
          "strcmp(s3, s1) = ", strcmp(s3, s1));

   printf("%s%2d\n%s%2d\n%s%2d\n",
          "strncmp(s1, s3, 6) = ", strncmp(s1, s3, 6),
          "strncmp(s1, s3, 7) = ", strncmp(s1, s3, 7),
          "strncmp(s3, s1, 7) = ", strncmp(s3, s1, 7));
   return 0;
}
```

```
s1 = Happy New Year
s2 = Happy New Year
s3 = Happy Holidays

strcmp(s1, s2) =  0
strcmp(s1, s3) =  1
strcmp(s3, s1) = -1

strncmp(s1, s3, 6) =  0
strncmp(s1, s3, 7) =  1
strncmp(s3, s1, 7) = -1
```

Рис. 8.21. Применение **strcmp** и **strncmp**

Совет по переносимости программ 8.2

Внутренние числовые коды, использующиеся для представления символов, могут отличаться на различных компьютерах.

В целях стандартизации представления символов большинство производителей компьютеров разрабатывают свои машины для работы с одной из двух популярных схем кодировки — *ASCII* или *EBCDIC*. ASCII — это аббревиатура «American Standard Code for Information Interchange» (Американский стандартный код для обмена информацией), а EBCDIC означает «Extended Binary Coded Decimal Interchange Code» (Расширенный двоично-десятичный код обмена информацией). Имеются и другие схемы кодирования, но эти две — наиболее популярны.

ASCII и EBCDIC называются *кодами символов* (*character codes*) или *наборами символов* (*characters sets*). Фактически операции со строками и символами выполняются не с самими символами, а с соответствующими числовыми кодами. Это объясняет взаимозаменяемость символов и небольших целых чисел в C. Следовательно, поскольку правомерно говорить, что один числовой код «больше», «меньше» или «равен» другому числовому коду, то же самое

становится правомерным в отношении различных символов или строк, если сравнивать их между собой посредством ссылки на соответствующие числовые коды. В приложении Г приведен список кодов символов стандарта ASCII.

8.8. Функции поиска из библиотеки обработки строк

В этом разделе представлены функции библиотеки обработки строк, использующиеся для поиска в строках символов и других строк (подстрок). Перечень этих функций приведен на рис. 8.22. Обратите внимание, что функции **strcspn** и **strspn** возвращают тип **size_t**. Тип **size_t** определяется как целочисленный тип, возвращаемый операцией **sizeof**.

Совет по переносимости программ 8.3

Тип **size_t** является системно-зависимым синонимом либо для типа **unsigned long**, либо для типа **unsigned int**.

Прототип	Описание функции
`char *strchr (const char *s, int c)`	
	Находит позицию первого вхождения символа **c** в строку **s**. Если **c** найден, функция возвращает указатель на **c** в строке **s**. В противном случае возвращается указатель со значением **NULL**.
`size_t strcspn (const char *s1, const char *s2)`	
	Определяет и возвращает длину начального сегмента строки **s1**, содержащего только те символы, которые не входят в строку **s2**.
`size_t strspn (const char *s1, const char *s2)`	
	Определяет и возвращает длину начального сегмента строки **s1**, содержащего только те символы, которые входят в строку **s2**.
`char *strprbk (const char *s1, const char *s2)`	
	Находит в строке **s1** позицию первого вхождения любого из символов строки **s2**. Если символ из строки **s2** найден, функция возвращает указатель на этот символ в строке **s1**. В противном случае возвращается указатель со значением **NULL**.
`char *strrchr (const char *s, int c)`	
	Находит позицию последнего вхождения символа **c** в строку **s**. Если **c** найден, функция возвращает указатель на **c** в строке **s**. В противном случае возвращается указатель со значением **NULL**.
`char *strstr (const char *s1, const char *s2)`	
	Находит позицию первого вхождения строки **s2** в строку **s1**. Если подстрока найдена, возвращает указатель подстроки в строке **s1**. В противном случае возвращается указатель со значением **NULL**.

Прототип	Описание функции
`char *strtok (char *s1, const char *s2)`	
	Последовательный вызов функции выполняет разбиение строки **s1** на лексемы (логические части, такие как слова в текстовой строке), разделенные символами, содержащимися в строке **s2**. При первом вызове функция получает в качестве аргумента строку **s1**, а в последующих вызовах, чтобы продолжить разбиение той же самой строки, в качестве первого аргумента передается **NULL**. При каждом вызове возвращается указатель на текущую лексему строки **s1**. Если при очередном вызове функция находит, что лексем в строке не осталось, то возвращается **NULL**.

Рис. 8.22. Функции операций со строками из библиотеки обработки строк

Функция ***strchr*** выполняет поиск заданного символа в строке до первого положительного результата (первого вхождения данного символа в строку) и возвращает указатель на этот символ. Если символ не найден, то **strchr** возвращает **NULL**. Программа, приведенная на рис. 8.23, использует **strchr** для поиска первого из символов '**a**' и первого из символов '**z**', входящих (или не входящих) в строку **"This is a test"**.

```
/* Применение strchr */
#include <stdio.h>
#include <string.h>

main()
{
   char *string = "This is a test";
   char character1 = 'a', character2 = 'z';

   if (strchr(string, character1) != NULL)
      printf("\'%c\' was found in \"%s\".\n",
             character1, string);
   else
      printf("\'%c\' was not found in \"%s\".\n",
             character1, string);
   if (strchr(string, character2) != NULL)
      printf("\'%c\' was found in \"%s\".\n",
             character2, string);
   else
      printf("\'%c\' was not found in \"%s\".\n",
             character2, string);
   return 0;
}

'a' was found in "This is a test".
'z' was not found in "This is a test".
```

Рис. 8.23. Применение **strchr**

Функция ***strcspn*** (рис. 8.24) определяет длину начальной части строки в первом аргументе, которая (часть) не содержит ни одного из символов, входящих в строку второго аргумента. Функция возвращает длину найденного сегмента.

```
/* Применение strcspn */
#include <stdio.h>
#include <string.h>

main()
{
   char *string1 = "The value is 3.14159";
   char *string2 = "1234567890";
   printf("%s%s\n%s%s\n\n%s\n%s%u",
          "string1 = ", string1, "string2 = ", string2,
          "The length of the initial segment of string1",
          "containing no characters from string2 = ",
          strcspn(string1, string2));
   return 0;
}
```

```
string1 = The value is 3.14159
string2 = 1234567890

The length of the initial segment of string1
containing no characters from string2 = 13
```

Рис. 8.24. Применение **strcspn**

Функция ***strpbrk*** выполняет поиск первого вхождения любого из символов строки второго аргумента в строку первого аргумента. Если ни один из символов второго аргумента не найден, **strpbrk** возвращает **NULL**. Программа, приведенная на рис. 8.25, выполняет поиск в строке **string1** первого по положению символа, который встречается в строке **string2**.

```
/* Применение strpbrk */
#include <stdio.h>
#include <string.h>

main()
{
   char *string1 = "This is a test";
   char *string2 = "beware";

   printf("%s\"%s\"\n'%c'%s\n\"%s\"\n",
          "Of the characters in ", string2,
          *strpbrk(string1, string2),
          " is the first character to appear in ", string1);
   return 0;
}
```

```
Of the characters in "beware"
'a' is the first character to appear in
"This is a test"
```

Рис. 8.25. Применение **strpbrk**

Функция ***strrchr*** выполняет поиск заданного символа в строке до последнего положительного результата (последнее вхождение данного символа в строку) и возвращает указатель на этот символ. Если символ не найден, то

strrchr возвращает **NULL**. Программа, приведенная на рис. 8.26, вызывает **strrchr** для поиска последнего из символов '**z**', входящих в строку "**A zoo has many animals including zebras**".

```
/* Применение strrchr */
#include <stdio.h>
#include <string.h>

main()
{
   char *string1 = "A zoo has many animals including zebras";
   int c = 'z' ;

   printf("%s\n%s'%c'%s\"%s\"\n",
          "The remainder of string1 beginning with the ",
          "last occurrence of character ", c,
          " is: ", strrchr (string1, c));
   return 0;
}
```

```
The remainder of string1 beginning with the
last occurrence of character 'z' is: "zebras"
```

Рис. 8.26. Применение **strrchr**

Функция ***strspn*** (рис. 8.27) определяет длину начальной части строки в первом аргументе, которая (часть) содержит только символы строки во втором аргументе. Функция возвращает длину найденного сегмента.

```
/* Применение strspn */
#include <stdio.h>
#include <string.h>

main()
{
   char *string1 = "The value is 3.14159";
   char *string2 = "aehilsTuv";

   printf("%s%s\n%s%s\n\n%s\n%s%u",
          "string1 = ", string1, "string2 = ", string2,
          "The length of the initial segment of string1",
          "containing only characters from string2 = ",
          strspn(string1, string2));
   return 0;
   }
```

```
string1 = The value is 3.14159
string2 = aehilsTuv

The length of the initial segment of string1
containing only characters from string2 = 13
```

Рис. 8.27. Применение **strspn**

Функция ***strstr*** выполняет поиск строки второго аргумента в строке первого аргумента до первого положительного результата (первого вхождения подстроки в строку первого аргумента). Если вторая строка найдена в первой строке, функция возвращает указатель на найденную подстроку в первом аргументе. Программа, приведенная на рис. 8.28, использует **strstr** для поиска подстроки "**def**" в строке "**abcdefabcdef**".

```
/* Применение strstr */
#include <stdio.h>
#include <string.h>

main()
{
   char *string1 = "abcdefabcdef";
   char *string2 = "def";

   printf("%s%s\n%s%s\n\n%s\n%s%s\n",
          "string1 = ", string1, "string2 = ", string2,
          "The remainder of string1 beginning with the ",
          "first occurrence of string2 is: ",
          strstr(string1, string2));
   return 0;
}
```

```
string1 = abcdefabcdef
string2 = def

The remainder of string1 beginning with the
first occurrence of string2 is: defabcdef
```

Рис. 8.28. Применение **strstr**

Функция ***strtok*** используется для разбиения строки на ряд лексем. Лексема — это последовательность символов, отделенная *разделительными символами* (обычно пробелами или знаками пунктуации). Например, в строке текста каждое слово можно рассматривать как лексему, а пробелы, отделяющие его от других слов, как разделительные символы.

Чтобы разбить строку на несколько лексем, требуется многократный вызов функции **strtok** (предполагаем, что строка содержит более одной лексемы). При первом вызове функция **strtok** получает два аргумента: строку, которую нужно разбить на лексемы, и строку, содержащую символы, использующиеся для разбиения. В программе, приведенной на рис. 8.29, оператор

```
tokenPtr = strtok (string, " ");
```

присваивает **tokenPtr** значение указателя на первую лексему в строке **string**. Второй аргумент функции **strtok** — " " — означает, что лексемы в **string** отделяются пробелами. Функция **strtok** выполняет поиск первого символа в **string**, который не является разделительным символом (пробелом). С этого символа начинается первая лексема. После этого функция ищет следующий разделительный символ в строке и заменяет его нулевым символом ('**\0**'). Этот символ завершает текущую лексему. Функция **strtok** запоминает значение указателя на символ, следующий сразу за найденной лексемой, и возвращает указатель на текущую лексему.

```
/* Применение strtok */
#include <stdio.h>
#include <string.h>

main ()
{
   char string[] = "This is a sentence with 7 tokens";
   char *tokenPtr;

   printf("%s\n%s\n\n%s\n",
          "The string to be tokenized is:", string,
          "The tokens are:");

   tokenPtr = strtok(string, " ");

   while (tokenPtr != NULL) {
      printf ("%s\n", tokenPtr);
      tokenPtr = strtok(NULL, " ");
   }
   return 0;
}
```

```
The string to be tokenized is:
This is a sentence with 7 tokens

The tokens are:
This
is
a
sentence
with
7
tokens
```

Рис. 8.29. Применение **strtok**

При последующих вызовах для продолжения разбиения той же строки функция **strtok** получает в качестве первого аргумента символ **NULL**. Аргумент **NULL** показывает, что при вызове **strtok** должна продолжить разбиение с того места строки **string**, которое было записано при последнем вызове функции. Если при очередном вызове **strtok** обнаруживается, что лексем уже не осталось, то функция возвращает **NULL**. Программа, приведенная на рис. 8.29, использует функцию **strtok** для разбиения на лексемы строки "**This is a sentence with 7 tokens**" ("**Это предложение содержит 7 лексем**"). Каждая лексема печатается отдельно. Отметьте, что **strtok** модифицирует входную строку и, следовательно, если строка будет использоваться в программе после обращения к функции **strtok**, то следует сделать копию исходной строки.

8.9. Функции памяти библиотеки обработки строк

Функции библиотеки обработки строк, представленные в этом разделе, упрощают выполнение операций с блоками памяти, сравнение блоков памяти и поиск в блоках памяти. Эти функции обращаются с блоками памяти, как с

массивами символов и могут оперировать с любыми блоками данных. На рис. 8.30 приведен перечень функций памяти библиотеки обработки строк. Далее в тексте, при обсуждении функций, термин «объект» обозначает блок данных.

Параметры-указатели для этих функций объявляются как **void ***. В главе 7 мы видели, что указатель любого типа данных может быть прямо присвоен указателю типа **void ***, и наоборот — указатель **void *** может прямо быть присвоен указателю на любой тип данных. Поэтому обсуждаемые функции могут получать в качестве параметра указатель на любой тип данных. Поскольку указатель типа **void *** не может быть разыменован (т.е. тип объекта, на который ссылается данный указатель, не может быть определен), каждая функция получает дополнительный аргумент, определяющий число символов (байтов), которое будет обрабатывать функция. Для простоты примеры этого раздела оперируют с массивами символов (блоками символов).

Прототип	Описание функции
`void *memcpy (void *s1, const void *s2, size_t n)`	
	Копирует **n** символов из объекта, указываемого **s2**, в объект, указываемый **s1**. Функция возвращает указатель, ссылающийся на результирующий объект.
`void *memmove (void *s1, const void *s2, size_t n)`	
	Копирует **n** символов из объекта, указываемого **s2**, в объект, указываемый **s1**. Копирование выполняется последовательно: символы из объекта, указываемого **s2**, копируются во временный массив и затем из временного массива копируются в объект, указываемый **s1**. Функция возвращает указатель, ссылающийся на результирующий объект.
`void *memcmp (const void *s1, const void *s2, size_t n)`	
	Сравнивает первые **n** символов объектов, указываемых **s1** и **s2**. Функция возвращает значения **0**, меньше **0** и больше **0**, если **s1** соответственно равен, меньше или больше чем **s2**.
`void *memchr (const void *s, int c, size_t n)`	
	Выполняет поиск первого вхождения **c** (преобразованного в тип **unsigned char**) в первых **n** символах объекта, указываемого **s**. Если **c** найден, возвращает указатель на **c** в объекте. В противном случае возвращает **NULL**.
`void *memset (const void *s, int c, size_t n)`	
	Копирует **c** (преобразованный в тип **unsigned char**) в первые **n** символов объекта, указываемого **s**. Возвращает указатель результата.

Рис. 8.30. Функции памяти из библиотеки обработки строк

Функция ***memcpy*** копирует определенное число символов из объекта, на который ссылается ее второй аргумент, в объект, на который ссылается первый аргумент. Функция может получать указатель на любой тип объекта. Поведение этой функции не определено в том случае, если два объекта перекрываются в памяти, т.е. являются частями одного и того же объекта. Программа, приведенная на рис. 8.31, использует ***memcpy*** для копирования строки, содержащейся в массиве **s2**, в массив **s1**.

```
/* Применение memcpy */
#include <stdio.h>
#include <string.h>

main ()
{
   char s1[22], s2[] = "Copy this string";

   memcpy (s1, s2, 17);
   printf("%s\n%s\"%s\"\n" ,
          "After s2 is copied into s1 with memcpy,",
          "s1 contains ", s1);
   return 0;
}
```

```
After s2 is copied into s1 with memcpy,
s1 contains "Copy this string"
```

Рис. 8.31. Применение **memcpy**

Функция ***memmove***, как и функция **memcpy**, копирует определенное число байтов из объекта, на который ссылается ее второй аргумент, в объект, на который ссылается первый аргумент. Сначала функция копирует заданное число байтов из второго аргумента во временный символьный массив, а затем копирует этот массив в первый аргумент. Такой способ позволяет копировать символы из одной части строки в другую часть той же самой строки.

Распространенная ошибка программирования 8.8

Функции операций со строками, за исключением функции **memmove**, дают неопределенный результат, когда копирование происходит между частями одной и той же строки.

Программа, приведенная на рис. 8.32, использует функцию **memmove** для копирования последних 10-ти байтов массива **x** в первые 10 байтов того же массива **x**.

```
/* Применение memmove */
#include <stdio.h>
#include <string.h>

main()
{
   char x[] = "Home Sweet Home";

   printf("%s%s\n",
          "The string in array x before memmove is: ", x);
   printf("%s%s\n",
          " The string in array x before memmove is: ",
          memmove(x, &x[5], 10));
return 0 ;
}
```

```
The string in array x before memmove is: Home Sweet Home
The string in array x before memmove is: Sweet Home Home
```

Рис. 8.32. Применение **memmove**

Функция ***memcmp*** (рис. 8.33) сравнивает определенное число символов первого аргумента с соответствующими символами второго аргумента. Функция возвращает значение больше 0, если первый аргумент больше второго аргумента, возвращает 0, если аргументы равны, и возвращает значение меньше 0, если первый аргумент меньше второго.

```
/* Применение memcmp */
#include <stdio.h>
#include <string.h>

main ()
{
   char s1[] = "ABCDEFG", s2[] = "ABCDXYZ";

   printf("%s%s\n%s%s\n\n%s%2d\n%s%2d\n%s%2d\n",
          "s1 = ", s1, "s2 = ", s2,
          "memcmp(s1, s2, 4) = ", memcmp(s1, s2, 4),
          "memcmp(s1, s2, 7) = ", memcmp(s1, s2, 7),
          "memcmp(s2, s1, 7) = ", memcmp(s2, s1, 7));
return 0;
}
```

```
s1 = ABCDEFG
s2 = ABCDXYZ

memcmp(s1, s2, 4) =  0
memcmp(s1, s2, 7) = -1
memcmp(s2, s1, 7) =  1
```

Рис. 8.33. Применение **memcpy**

Функция ***memchr*** выполняет поиск байта (до первого положительного результата), представленного типом **unsigned char**, в заданном числе байтов объекта. Если заданный байт найден, функция возвращает указатель на найденный байт объекта. В противном случае функция возвращает указатель со значением **NULL**. Программа, приведенная на рис. 8.34, выполняет поиск символа (байта) '**r**' в строке "**This is a string**".

```
/* Применение memchr */
#include <stdio.h>
#include <string.h>

main()
{
   char *s = "This is a string";

   printf("%s\'%c\'%s\"%s\"\n",
          "The remainder of s after character ", 'r',
          " is found is ", memchr(s, 'r', 16));
   return 0;
}
```

```
The remainder of s after character 'r' is "ring"
```

Рис. 8.34. Применение **memchr**

Функция ***memset*** копирует значение байта второго аргумента в определенное число байтов объекта, на который указывает первый аргумент. Программа, приведенная на рис. 8.35, использует функцию **memset** для копирования **'b'** в первые **7** байтов **string1**.

```
/* Применение memset */
#include <stdio.h>
#include <string.h>

main()
{
   char string1[15] = "BBBBBBBBBBBBBB";

   printf("string1 = %s\n", string1);
   printf("string1 после memset = %s\n",
          memset(string1, 'b', 7));
   return 0;
}
```

```
string1 = BBBBBBBBBBBBBB
stringi после memset = bbbbbbbBBBBBBB
```

Рис. 8.35. Применение **memset**

8.10. Другие функции из библиотеки обработки строк

Двумя последними функциями библиотеки обработки строк являются **strerror** и **strlen**. Эти функции приведены на рис. 8.36.

Прототип	Описание функции
`int strerror (int errornum)`	Устанавливает соответствие номера ошибки **errornum** полной текстовой строке (зависящей от системы). Возвращает указатель на строку.
`int strlen (const char *s)`	Определяет длину строки **s**. Возвращает число символов, предшествующих ограничивающему символу **NULL**.

Рис. 8.36. Функции операций со строками из библиотеки обработки строк

Функция ***strerror*** получает номер ошибки и создает строку сообщения об ошибке. Функция возвращает указатель на эту строку. Программа, приведенная на рис. 8.37, демонстрирует работу **strerror**.

Совет по переносимости программ 8.4

Сообщение об ошибке, генерируемое **strerror**, является системно-зависимым.

Функция ***strlen*** получает строку в качестве аргумента и возвращает число символов в строке (ограничивающий символ **NULL** в длину строки не включен). Программа, приведенная на рис. 8.38, демонстрирует работу **strlen**.

```
/* Применение strerror */
#include <stdio.h>
#include <string.h>

main()
{
   printf("%s\n", strerror(2));
   return 0;
}
```

```
Error 2
```

Рис. 8.37. Применение **strerror**

```
/* Применение strlen */
#include <stdio.h>
#include <string.h>

main ()
{
   char *string1 = "abcdefghijklmnopqrstuvwxyz";
   char *string2 = "four";
   char *string3 = "Boston";

   printf("%s\"%s\"%s%lu\n%s\"%s\"%s%lu\n%s\"%s\"%s%lu\n",
          "The length of ", string1, " is ", strlen(string1),
          "The length of ", string2, " is ", strlen(string2),
          "The length of ", string3, " is ", strlen(string3));
   return 0;
}
```

```
The length of "abcdefghijklmnopqrstuvwxyz" is 26
The length of "four" is 4
The length of "Boston" is 6
```

Рис. 8.38. Применение **strlen**

Резюме

- Функция **islower** определяет, является ли ее аргумент буквой нижнего регистра **(a–z)**.
- Функция **isupper** определяет, является ли ее аргумент буквой верхнего регистра **(A–Z)**.
- Функция **isdigit** определяет, является ли ее аргумент цифрой **(0–9)**.
- Функция **isalpha** определяет, попадает ли ее аргумент в список букв верхнего **(A–Z)** или нижнего **(a–z)** регистров.
- Функция **isalnum** определяет принадлежность аргумента буквам верхнего и нижнего регистров или цифрам.
- Функция **isxdigit** определяет, является ли аргумент шестнадцатеричной цифрой **(A–F, a–f, 0–9)**.
- Функция **toupper** преобразует регистр буквы из нижнего в верхний и возвращает букву верхнего регистра.

- Функция **tolower** преобразует регистр буквы из верхнего в нижний и возвращает букву нижнего регистра.
- Функция **isspace** определяет принадлежность аргумента следующему списку пробельных символов: (' '), ('\f'), ('\n'), ('\r'), ('\t'), ('\v').
- Функция **iscntrl** определяет принадлежность аргумента следующему списку управляющих символов: ('\t'), ('\v'), ('\f'), ('\a'), ('\b'), ('\r'), ('\n').
- Функция **ispunct** определяет, является ли аргумент отображаемым символом, не относящимся к цифрам, буквам или пробелу.
- Функция **isprint** определяет, является ли аргумент отображаемым символом, включая символ пробела.
- Функция **isgraph** определяет, является ли аргумент отображаемым символом, исключая символ пробела.
- Функция **atof** преобразует аргумент — строку, которая начинается с ряда цифр, представляющих число с плавающей точкой, — в значение типа **double**.
- Функция **atoi** преобразует аргумент — строку, которая начинается с ряда цифр, представляющих целое число, — в значение типа **int**.
- Функция **atol** преобразует аргумент — строку, которая начинается с ряда цифр, представляющих длинное целое (**long int**) — в значение типа **long**.
- Функция **strtod** преобразует последовательность символов, представляющих число с плавающей точкой, в значение типа **double**. Эта функция получает два аргумента: строку (**char ***) и указатель на (**char ***). Строка содержит последовательность символов, которые должны быть преобразованы, а указатель (**char ***) после преобразования ссылается на строку-остаток.
- Функция **strtol** преобразует последовательность символов, представляющих целое число, в значение типа **long**. Эта функция получает три аргумента: строку (**char ***), указатель на **char *** и целое число. Строка содержит последовательность символов, которые должны быть преобразованы. Указатель (**char ***) после преобразования ссылается на строку-остаток. Целое число определяет основание, по которому проводится преобразование.
- Функция **strtoul** преобразует последовательность символов, представляющих целое число, в значение типа **unsigned long**. Эта функция получает три аргумента: строку (**char ***), указатель на **char *** и целое число. Строка содержит последовательность символов, которые должны быть преобразованы. Указатель (**char ***) после преобразования ссылается на строку-остаток. Третий аргумент определяет основание, по которому проводится преобразование.
- Функция **gets** читает символы со стандартного входного устройства (клавиатуры) до тех пор, пока не встретит символ новой строки или индикатор конца файла. Аргументом **gets** является массив типа **char**. После окончания считывания в массив добавляется символ **NULL** ('\0').
- Функция **putchar** печатает свой аргумент — символ.

- Функция **getchar** читает символ со стандартного входного устройства и возвращает целое значение символа. Если встречается индикатор конца файла, **getchar** возвращает **EOF**.
- Функция **puts** получает в качестве аргумента строку (**char ***) и выводит ее с последующим символом новой строки.
- Функция **sprintf** использует ту же самую спецификацию преобразования, что и **printf** и выполняет форматированный вывод в массив типа **char**.
- Функция **sscanf** использует ту же самую спецификацию преобразования, что и **scanf** и читает форматированные данные из строки.
- Функция **strcopy** копирует второй аргумент — строку — в первый аргумент — массив символов. Программист должен следить за размером массива, который должен иметь достаточный размер, чтобы запомнить строку и ограничивающий символ **NULL**.
- Функция **strncopy** эквивалентна **strcopy** за исключением того, что **strncopy** задает число символов, которые необходимо скопировать из строки в массив. Ограничивающий символ **NULL** будет скопирован, только если число копируемых символов превышает длину строки по меньшей мере на единицу.
- Функция **strcat** выполняет объединение строк (конкатенацию): второго аргумента — строки, включая ограничивающий символ **NULL**, — с первым аргументом — массивом символов, содержащим строку. Первый символ второго аргумента заменяет последний символ **NULL** (**'\0'**) строки первого аргумента. Программист должен убедиться, что массив, который содержит первую строку, достаточен по размерам для хранения обеих строк.
- Функция **strncat** выполняет объединение определенного числа символов из второй строки с первой строкой. К результирующей строке добавляется ограничивающий символ **NULL**.
- Функция **strcmp** сравнивает первый аргумент-строку со вторым аргументом-строкой символ за символом. Функция возвращает **0**, если строки равны, отрицательное значение, если первая строка меньше второй, и положительное значение, если первая строка больше второй.
- Функция **strncmp** эквивалентна функции **strcmp** за исключением того, что **strncmp** сравнивает заданное число символов в строках. Если число символов в одной из строк меньше чем заданное число, **strncmp** сравнивает символы до тех пор, пока не встретит ограничивающий символ **NULL** в более короткой строке.
- Функция **strchr** выполняет поиск первого вхождения данного символа в строку и возвращает указатель на этот символ. Если символ не найден, **strchr** возвращает **NULL**.
- Функция **strcspn** определяет длину начальной части строки в первом аргументе, которая (часть) не содержит ни одного из символов, входящих в строку второго аргумента. Эта функция возвращает длину найденного сегмента.

- Функция **strpbrk** выполняет поиск первого вхождения любого из символов строки второго аргумента в строку первого аргумента и возвращает указатель на этот символ. Если ни один из символов второго аргумента не найден, **strpbrk** возвращает **NULL**.
- Функция **strrchr** выполняет поиск последнего вхождения данного символа в строку и возвращает указатель на этот символ. Если символ не найден, то **strchr** возвращает **NULL**.
- Функция **strspn** определяет длину начальной части строки в первом аргументе, которая (часть) содержит только те символы, которые входят в строку во втором аргументе. Эта функция возвращает длину найденного сегмента.
- Функция **strstr** выполняет поиск первого вхождения строки второго аргумента в строку первого аргумента. Если вторая строка найдена в первой строке, функция возвращает указатель на найденную подстроку в первом аргументе.
- Последовательный вызов функции **strtok** выполняет разбиение строки **s1** на лексемы, разделенные символами, содержащимися в строке **s2**. При первом вызове функция получает в качестве аргумента строку **s1**, а в последующих вызовах, чтобы продолжить разбиение той же самой строки, в качестве первого аргумента предается **NULL**. При каждом вызове возвращается указатель на текущую лексему. Если при очередном вызове функция находит, что лексем в строке не осталось, то возвращается **NULL**.
- Функция **memcpy** копирует заданное число символов из объекта, на который ссылается ее второй аргумент, в объект, на который ссылается первый аргумент. Функция может получать указатель на любой тип объекта. Функция **memcpy** получает указатель как указатель типа **void**, который, для использования в функции, преобразуется в указатель типа **char**. Функция memcpy оперирует с байтами объектов как с символами.
- Функция **memmove** копирует заданное число байтов из объекта, на который ссылается ее второй аргумент, в объект, на который ссылается первый аргумент. Сначала функция копирует заданное число байтов из второго аргумента во временный символьный массив, а затем копирует этот массив в первый аргумент.
- Функция **memcmp** сравнивает заданное число символов первого и второго аргументов.
- Функция **memchr** выполняет поиск байта (до первого положительного результата), представленного типом **unsigned char**, в заданном числе байтов объекта. Если заданный байт найден, функция возвращает указатель на найденный байт объекта. В противном случае функция возвращает указатель со значением **NULL**.
- Функция **memset** копирует второй аргумент, который обрабатывается как **unsigned char**, в заданное число байтов объекта, на который указывает первый аргумент.

- Функция **strerror** устанавливает соответствие номера ошибки **errornum** полной текстовой строке (зависящей от системы). Возвращает указатель на строку.
- Функция **strlen** определяет длину строки — аргумента и возвращает число символов в строке; ограничивающий символ **NULL** не включается в длину строки.

Терминология

ASCII
atof
atoi
atol
ctype.h
EBCDIC
getchar
gets
isalnum
isalpha
iscntrl
isdigit
isgraph
islower
isprint
ispunct
isspace
isupper
isxdigit
memchr
memcmp
memcpy
memmove
memset
putchar
puts
sprintf
sscanf
stdio.h
stdlib.h
strcat
strchr
strcmp
strcpy
strcspn
strerror
string.h
strlen
strncat
strncmp
strncpy
strpbrk
strrchr
strspn
strstr
strtod
strtok
strtol
strtoul
tolower
toupper
библиотека утилит общего назначения
длина строки
код символа
конкатенация строк
копирование строк
лексема
литерал
набор символов
обработка строк
обработка текстов
объединение строк с другими строками
отображаемый символ
представление символа числовым кодом
пробельные символы
разбиение строки на лексемы
разделительные символы
символьная константа
сравнение строк
строка
строка поиска
строковая константа
строковый литерал
управляющий символ
функции преобразования строк
функции сравнения строк
шестнадцатеричные цифры

Распространенные ошибки программирования

8.1. При определении размера массива символов не выделено место, чтобы запомнить символ **NULL**, который ограничивает строку.

8.2. Попытка напечатать «строку», которая не содержит ограничивающего символа **NULL**.

8.3. Обработка одиночного символа в качестве строки. Строка — это указатель, который, возможно, представляется данными типа **long int**. Символ является небольшим целым числом (значения кодовой таблицы ASCII находятся в диапазоне 0-255). На многих системах такая замена вызовет ошибку, поскольку младшие адреса памяти резервируются для специального использования — например, для адресации программ обработки прерываний операционной системы, — так что произойдет «несанкционированный доступ».

8.4. Передача символа в качестве аргумента функции, которая оперирует со строками.

8.5. Передача строки в качестве аргумента функции, которая оперирует с символами.

8.6. Отсутствует ограничивающий символ **NULL** в первом аргументе функции **strncpy**, когда величина третьего аргумента меньше или равна длине строки во втором аргументе.

8.7. Предположение, что **strcmp** и **strncmp** возвращают 1 в случае равенства строк. На самом деле функции **strcmp** и **strncmp** возвращают в этом случае 0 (значение false в C) и, следовательно, при проверке двух строк на равенство результат должен сравниваться с 0.

8.8. Функции операций со строками, за исключением функции **memmove**, дают неопределенный результат, когда копирование происходит между частями одной и той же строки.

Хороший стиль программирования

8.1. Когда вы объявляете массив символов для хранения строки, убедитесь, что размер его достаточен для хранения самой большой строки, с которой предполагается работать. C позволяет запоминать строку любой длины. Если строка оказалась длиннее массива, в котором она должна сохраняться, то символы, выходящие за размер массива, перепишут данные в области памяти, следующей сразу за массивом.

8.2. Если вы вызываете функции библиотеки обработки символов, подключите заголовочный файл **<ctype.h>**.

8.3. Если вы вызываете функции из библиотеки утилит общего назначения, подключите заголовочный файл **<stlib.h>**.

8.4. Если вы вызываете функции из стандартной библиотеки ввода/вывода, подключите заголовочный файл **<stdio.h>**.

8.5. Если вы вызываете функции из библиотеки обработки строк, подключите заголовочный файл **<string.h>**.

Советы по переносимости программ

8.1. Если переменная типа **char *** инициализируется строкой символьных констант, то некоторые компиляторы могут поместить такую

строку в неизменяемую область памяти. Если, например, вам требуется изменять какие-то символы в литеральной строке, то, чтобы иметь такую возможность на всех системах, строку нужно запомнить в массиве символов.

8.2. Внутренние числовые коды, использующиеся для представления символов, могут отличаться на различных компьютерах.

8.3. Тип **size_t** является системно-зависимым синонимом либо для типа **unsigned long**, либо для типа **unsigned int**.

8.4. Содержание сообщения, генерируемого **strerror**, является системно-зависимым.

Упражнения для самоконтроля

8.1. Предположим, что переменная **c** (содержащая символ), **x**, **y**, и **z** являются переменными типа **int**, переменные **d**, **e**, и **f** являются переменными с плавающей точкой, переменная **ptr** имеет тип **char *** и массивы **s1 [100]** и **s2 [100]** являются массивами типа **char**. Напишите по одному оператору, чтобы выполнить каждое из следующих заданий.

a) Преобразуйте символ, хранящийся в переменной **c**, в букву верхнего регистра. Присвойте результат переменной **c**.

b) Определите, является ли значение переменной **c** цифрой. Используйте условную операцию, как показано на рис. 8.2, 8.3, и 8.4, чтобы при выводе результата печатались фразы **«is a»** или **«is not a»**.

c) Преобразуйте строку **«1234567»** в тип **long** и выведите полученное значение.

d) Определите, является ли значение переменной с управляющим символом. Используйте условную операцию, чтобы при выводе результата печатались фразы **«is a»** или **«is not a»**.

e) Введите с клавиатуры строку текста в массив **s1** не используя функцию **scanf**.

f) Выведите строку текста, хранящуюся в массиве **s1**, не используя **printf**.

g) Присвойте **ptr** местоположение последнего вхождения символа **c** в **s1**.

h) Выведите значение переменной **c** не используя **printf**.

i) Преобразуйте строку **«8.63582»** в тип **double** и выведите полученное значение.

j) Определите, является ли значение **c** буквой. Используйте условную операцию, чтобы при выводе результата печатались фразы **«is a»** или **«is not a»**.

k) Введите с клавиатуры символ и сохраните его в переменной **c**.

1) Присвойте **ptr** местоположение первого вхождения **s2** в **s1**.

m) Определите, является ли значение переменной **c** печатаемым символом. Используйте условную операцию, чтобы при выводе результата печатались фразы **«is a»** или **«is not a»**.

n) Введите три значения типа **float** в переменные **d**, **e**, и **f** из строки **«1.27 10.3 9.432»**.

o) Скопируйте строку, хранящуюся в массиве **s2**, в массив **s1**.

p) Присвойте **ptr** местоположение первого вхождения в **s1** любого символа из **s2**.

q) Сравните строку в **s1** со строкой в **s2**. Выведите результат.

r) Присвойте **ptr** местоположение первого вхождения **c** в **s1**.

s) Используйте **sprintf** для вывода значений целых переменных **x**, **y**, и **z** в массив **s1**. Каждое значение должно быть напечатано с шириной поля, равной **7**.

t) Выполните конкатенацию 10 символов строки в **s2** со строкой в **s1**.

u) Определите длину строки в **s1** и выведите результат.

v) Преобразуйте строку **«-21»** в тип **int** и выведите полученное значение.

w) Присвойте **ptr** местоположение первой лексемы в **s2**. Лексемы в **s2** отделены запятыми (,).

8.2. Покажите два различных метода инициализации массива символов **vowel** строкой гласных **«AEIOU»**.

8.3. Что выводится (если вообще выводится) при выполнении каждого из следующих операторов С? Если оператор содержит ошибку, охарактеризуйте ее и укажите, как ее исправить. Предположим, что объявлены следующие переменные:

```
char s1[50] = "jack", s2[50] = "jill", s3[50], *sptr;
```

a) `printf("%c%s", toupper(s1[0]), &s1[1]);`

b) `printf("%s", strcpy(s3, s2));`

c) `printf("%s", strcat(strcat(strcpy(s3, s1), " and "), s2));`

d) `printf("%u", strlen(s1) + strlen(s2));`

e) `printf("%u", strlen(s3));`

8.4. Найдите ошибку в каждом из следующих операторов программы и объясните, как их исправить:

a)
```
char s[10];
strncpy(s, "hello", 5);
printf("%s\n", s);
```

b) `printf("%s", 'a');`

c)
```
char s[12];
strcpy(s, "Welcome Home");
```

d)
```
if (strcmp(string1, string2))
   printf("The strings are egual\n");
```

Ответы к упражнениям для самоконтроля

8.1. a) `c = toupper(c);`

b)
```
printf("'%c'%sdigit\n",
        c, isdigit(c) ? " is a " : " is not a ");
```

c) `printf("%ld\n", atol("1234567"));`

d)
```
printf("'%c'%scontrol character\n",
       c, iscntrl(c) ? " is a " : " is not a ");
```

e) `gets(s1);`

f) `puts(s1);`

g) `ptr = strrchr(s1, c);`

h) `putchar(c);`

i) `printf("%f\n", atof("8.63582"));`

j)
```
printf("'%c'%sletter\n",
       c, isalpha(c) ? " is a " : " is not a ");
```

k) `c = getchar();`

l) `ptr = strstr(s1, s2);`

m)
```
printf("'%c'%sprinting character\n",
       c, isprint(c) ? " is a " : " is not a ");
```

n) `sscanf("1.27 10.3 9.432", "%f%f%f", &d, &e, &f);`

o) `strcpy(s1, s2);`

p) `ptr = strpbrk(s1, s2);`

q) `printf("strcmp(s1, s2) = %d\n", strcmp(s1, s2));`

r) `ptr = strchr(s1, c);`

s) `sprintf(s1, "%7d%7d%7d", x, y, z);`

t) `strncat(s1, s2, 10);`

u) `printf("strlen(s1) = %u\n", strlen(s1));`

v) `printf("%d\n", atoi("-21"));`

w) `ptr = strtok(s2, ",");`

8.2.
```
char vowel[] = "AEIOU";
char vowel[] = {'A', 'E', 'I', 'O', 'U', '\0'};
```

8.3. a) Jack

b) jill

c) jack and jill

d) 8

e) 13

8.4. a) Ошибка: функция **strncpy** не записывает ограничивающий символ **NULL** в массив **s**, поскольку ее третий аргумент равен длине строки **«hello»**.

Исправление: сделайте третий аргумент функции **strncpy** равным **6** или присвойте **'\0'** элементу массива **s [5]**.

b) Ошибка: попытка вывести символьную константу как строку.

Исправление: используйте формат **%c** для вывода символа или замените **'a'** на **"a"**.

c) Ошибка: массив символов **s** имеет недостаточный размер, чтобы уместить ограничивающий символ **NULL**.

Исправление: объявите массив с большим количеством элементов.

d) Ошибка: в случае равенства строк функция **strcmp** возвращает 0 и, следовательно, функция **printf** в структуре **if** не будет выполнена.

Исправление: в условном операторе сравните с **0** значение, возвращаемое функцией **strcmp**.

Упражнения

8.5. Напишите программу, которая вводит символы с клавиатуры и проверяет их с помощью каждой из функций библиотеки обработки символов. Программа должна выводить значения, возвращенные каждой функцией.

8.6. Напишите программу, которая вводит строки текста в массив символов **s [100]**, используя функцию **gets**. Выведите строки в верхнем и нижнем регистрах.

8.7. Напишите программу, которая вводит 4 строки, представляющие целые значения, преобразует строки в целые числа, суммирует и выводит сумму 4 значений.

8.8. Напишите программу, которая вводит 4 строки, представляющие значения с плавающей точкой, преобразует строки, удваивая значения, суммирует и выводит сумму 4 значений.

8.9. Напишите программу, которая использует функцию **strcmp** для сравнения двух строк, введенных пользователем. Программа должна определить, является ли первая строка меньшей, равной или большей второй строки.

8.10. Напишите программу, которая использует функцию **strcmp** для сравнения двух строк, введенных пользователем. Программа должна вводить число сравниваемых символов. Программа должна определить, является ли первая строка меньшей, равной или большей второй строки.

8.11. Напишите программу, которая использует генератор случайных чисел для создания предложений. Программа должна использовать четыре массива указателей на **char** с именами **article (артикль)**, **noun (существительное)**, **verb (глагол)** и **preposition (предлог)**. Программа должна создавать предложения, выбирая случайным образом слова из массивов в следующем порядке: **article**, **noun**, **verb**, **preposition**, **article** и **noun**. После выбора очередного слова выполняется конкатенация с предыдущим словом в массиве, который должен быть достаточно большим, чтобы уместить полное предложение. Слова должны отделяться пробелами. Заключительное предложение при выводе должно начинаться с заглавной буквы и оканчиваться точкой. Программа должна образовать 20 таких предложений.

В массивах должны содержаться следующие слова: **article** должен содержать артикли **«the»**, **«a»**, **«one» (один)**, **«some» (некоторые)**, и **«any» (любой)**; **noun** должен содержать существительные **«boy» (юноша)**, **«girl» (девушка)**, **«dog» (собака)**, **«town» (город)** и **«car» (автомобиль)**; **verb** должен содержать глаголы **«drove» (ехал)**,

«jumped» (подпрыгнул), «ran» (побежал), «walked» (шел) и **«skipped» (скакал)**; **preposition** должен содержать предлоги **«to» (к), «from» (от), «over» (через)** и **«on» (на)**.

Когда программа будет написана и отлажена, измените ее так, чтобы она создала короткий рассказ, состоящий из нескольких таких предложений.

8.12. (*Лимерики*) Лимерик — юмористическое стихотворение из пяти строк, в котором первая и вторая строки рифмуются с пятой, а третья рифмуется с четвертой. Используя методику, рассмотренную в упражнении 8.11, напишите программу, которая производит случайные стихотворения. Отладка программы до уровня, на котором она создавала бы хорошие стихотворения, является серьезной и увлекательной проблемой, но результат будет стоить усилий!

8.13. Напишите программу, которая кодирует английские фразы во фразы «свинячьей латыни». Свинячья латынь — это форма кодированного языка, который часто используется в развлекательных целях. Существует много вариаций методов формирования фраз. Для простоты используйте следующий алгоритм.

Для формирования фразы свинячьей латыни из фразы английского языка разбейте последнюю на слова (лексемы), используя функцию **strtok**. Чтобы перевести каждое английское слово в слово свинячьей латыни, поместите первый символ слова в конец и добавьте буквы **«ay»**. Таким образом, слово **«jump»** становится **«umpjay»**, слово **«the»** становится словом **«hetay»** и слово **«computer»** становится **«omputercay»**. Пробелы между словами остаются пробелами. Предположим следующее: английская фраза состоит из слов, разделенных пробелами, отсутствуют любые знаки препинания и все слова имеют два или большее число символов. Функция **printLatinWord** должна выводить каждое слово. Подсказка: каждый раз, когда при обращении к функции **strtok** найдена очередная лексема, передайте указатель лексемы функции **printLatinWord** и выведите получившееся слово.

8.14. Напишите программу, которая вводит телефонный номер как строку в формате **(555) 555-5555**. Программа должна использовать функцию **strtok**, чтобы извлекать в качестве лексем код региона, первые три цифры номера телефона и последние четыре цифры номера телефона. Семь цифр номера телефона должны объединяться в одну строку операцией конкатенации. Программа должна преобразовать строку кода региона в тип **int**, а строку номера телефона в тип **long**. И код региона и номер должны быть выведены на печать.

8.15. Напишите программу, которая вводит строку текста, разбивает ее на лексемы с помощью функции **strtok** и выводит лексемы в обратном порядке.

8.16. Напишите программу, которая считывает с клавиатуры строку текста и строку поиска. Вызвав функцию **strstr**, найдите местоположение первого вхождения строки поиска в строку текста и присвойте ее местоположение переменной **searchPtr** типа **char ***. Если строка поиска найдена, выведите остаток строки текста, начинающийся со

строки поиска. Затем снова вызовите **strstr**, чтобы найти следующее вхождение строки поиска в строку текста. Если второе вхождение найдено, выведите остаток строки текста, начинающийся со второго вхождения строки поиска. Подсказка: второе обращение к **strstr** должно содержать в качестве первого аргумента значение **searchPtr+1**.

8.17. Напишите программу, основанную на программе упражнения 8.16, которая вводит несколько строк текста и строку поиска и использует функцию **strstr**, чтобы определить суммарное число вхождений строки поиска в текст. Выведите результат.

8.18. Напишите программу, которая вводит несколько строк текста и символ поиска и использует функцию **strchr**, чтобы определить суммарное число вхождений символа в текст.

8.19. Напишите программу, основанную на программе упражнения 8.18, которая вводит несколько строк текста и использует функцию **strchr**, чтобы определить суммарное количество вхождений каждой буквы алфавита в текст. Буквы верхнего и нижнего регистров должны суммироваться вместе. Сохраните в массиве общее число вхождений каждого символа и выведите эти значения в форме таблицы.

8.20. Напишите программу, которая вводит несколько строк текста и использует функцию **strtok**, чтобы сосчитать общее количество слов. Предположим, что слова разделяются символами новой строки или пробелами.

8.21. Используйте функции сравнения строк, рассмотренные в разделе 8.6, и методы сортировки массивов, развитые в главе 6, чтобы написать программу алфавитной сортировки списка строк. В качестве базы данных используйте названия 10 или 15 городов вашей области.

8.22. Таблица, приведенная в приложении Г, показывает численное представление символов в коде ASCII. Исследуйте эту таблицу и установите, является ли каждое из следующих утверждений верным или неверным.

a) Символ **«A»** находится перед символом **«B»**.

b) Цифра **9** находится перед цифрой **0**.

c) Все символы, которые обычно используются для сложения, вычитания, умножения и деления, находятся перед любой из цифр.

d) Цифры находятся перед символами.

e) Если программа сортирует строки в порядке возрастания значений, то она поместит символ правой круглой скобки перед символом левой круглой скобки.

8.23. Напишите программу, которая вводит ряд строк и выводит те из них, которые начинаются с буквы **«b»**.

8.24. Напишите программу, которая вводит ряд строк и выводит те из них, которые заканчиваются на **«ED»**.

8.25. Напишите программу, которая вводит код ASCII и выводит соответствующий символ. Измените эту программу так, чтобы она генерировала все возможные коды с тремя цифрами в диапазоне от 000 до 255 и попыталась вывести соответствующие символы. Что происходит при выполнении этой программы?

8.26. Используя таблицу символов ASCII приложения Г как руководство, напишите свои собственные версии функций обработки символов, приведенных на рис. 8.1.

8.27. Напишите свои собственные версии функций преобразования строк в числа, которые приведены на рис. 8.5.

8.28. Напишите по две версии функций копирования строки и конкатенации строк, приведенных на рис. 8.17. Первая версия должна использовать индексацию массива, а вторая версия должна использовать указатели и арифметические операции над указателями.

8.29. Напишите ваши собственные версии функций **getchar**, **gets**, **putchar** и **puts**, показанных на рис. 8.12.

8.30. Напишите две версии для каждой из функций сравнения строк, приведенных на рис. 8.20. Первая версия должна использовать индексацию массива, а вторая версия должна использовать указатели и арифметические операции над указателями.

8.31. Напишите ваши собственные версии функций поиска строк, приведенных на рис. 8.22.

8.32. Напишите ваши собственные версии функций управления блоками памяти, приведенных на рис. 8.30.

8.33. Напишите две версии функции **strlen**, приведенной на рис. 8.36. Первая версия должна использовать индексацию массива, а вторая должна использовать указатели и арифметические операции над указателями.

Специальный раздел: более сложные упражнения по работе со строками

Предыдущие упражнения важны для понимания текста главы и разработаны, чтобы проверить понимание читателем основных принципов, применяемых при обработке строк. Данный раздел включает рад задач средней и повышенной сложности. Читатель может найти некоторые из них чересчур сложными, однако их решение должно принести определенное удовлетворение. Задачи существенно различаются по сложности. Некоторые требуют всего одного или двух часов для написания и отладки программы. Другие полезны для лабораторных занятий; для их решения могут потребоваться две или три недели. Некоторые задачи могут послужить основой курсового проекта.

8.34. (*Анализ текста*) Доступность компьютеров с возможностями обработки строк сделала реальными довольно интересные идеи относительно анализа текстов великих авторов. Много внимания было сосредоточено на том, жил ли когда-нибудь Вильям Шекспир. Некоторые ученые полагают, что имеются достоверные указания на то, что ав-

тором шедевров, приписываемых Шекспиру, является Кристофер Марло. Исследователи применяли компьютеры, чтобы найти совпадения в текстах эти двух авторов. Данное упражнение исследует три метода компьютерного анализа текстов.

a) Напишите программу, которая читает несколько строк текста и выводит таблицу, показывающую число вхождений каждого символа алфавита в текст. Например, фраза

```
To be, or not to be: that is the question:
```

содержит один символ «a», два «b», ни одного «c» и т.д.

b) Напишите программу, которая читает несколько строк текста и выводит таблицу, показывающую число слов в тексте, содержащих один символ, два символа, три символа и т.д. Например, фраза

```
Whether 'tis nobler in the mind to suffer
```

содержит

Длина слова	Число слов
1	0
2	2
3	2
4	2 (включая 'tis)
5	0
6	2
7	1

c) Напишите программу, которая читает несколько строк текста и выводит таблицу, показывающую число вхождений каждого слова в текст. Первый вариант вашей программы должен включать слова в таблицу в том же самом порядке, в котором они появляются в тексте. Более интересной (и полезной) является таблица, в которой слова сортируются в алфавитном порядке. Например, строки

```
To be, or not to be: that is the question:
Whether 'tis nobler in the mind to suffer
```

содержит три слова «to», два слова «be», одно слово «or» и т.д.

8.35. (*Обработка текстов*) Детальное рассмотрение обработки строк в этой главе в значительной степени обязано захватывающему росту в последнее время числа приложений обработки текстов. Одной из важных функций в системах обработки слов является функция *выравнивания* — выравнивание слов по левому и правому полям страницы. Она генерирует профессионально выглядящий документ, который производит впечатление, что его скорее напечатали в типографии, чем подготовили на пишущей машинке. Выравнивание может быть выполнено на компьютерной системе, если вставить один или большее количество символов пробела между словами в строке таким образом, чтобы самое правое слово выравнивалось по правому полю.

Напишите программу, которая читает несколько строк текста и печатает этот текст в выровненном формате. Предположим, что текст должен быть напечатан на бумаге шириной 8 1/2 дюймов, а с левой и правой стороны напечатанной страницы должны оставаться поля шириной в один дюйм. Предположим также, что компьютер печатает 10 символов на дюйм по горизонтали. Следовательно, ваша программа должна выводить текст шириной 6 1/2 дюймов, что соответствует 65 символам в строке.

8.36. (*Печать дат в различных форматах*) Даты в деловой корреспонденции печатаются, как правило, в нескольких различных форматах. Двумя из наиболее распространенных форматов являются:

```
07/21/55 и July 21, 1955
```

Напишите программу, которая читает дату в первом формате и выводит во втором.

8.37 (*Защита чеков*) Компьютеры часто используются в системах печати чеков, например, при расчете зарплат и оплате счетов. Циркулирует много странных историй, относящихся к ошибочной выдаче чеков еженедельных зарплат на суммы более одного миллиона долларов. Сверхъестественные суммы напечатаны автоматизированными системами из-за ошибки человека и/или из-за отказа машины. Проектировщики, конечно, прилагают все усилия, чтобы встроить в свои системы средства контроля, предотвращающие ошибочную выдачу чеков.

Другая серьезная проблема — намеренное дописывание суммы чека мошенником, который предполагает получить наличные деньги по подложному чеку. Чтобы предотвратить дописывание суммы, большинство автоматизированных денежных систем используют методику, которая называется *защитой чеков*.

Чеки, разработанные для компьютерного вывода, содержат фиксированное число пробелов, в которых компьютер может печатать сумму. Предположим, что платежный чек содержит восемь пустых полей, в которых компьютер печатает еженедельную сумму выплаты. Если сумма большая, то будут заполнены все восемь полей, например:

```
1,230.60 (сумма чека)
--------
12345678 (позиции цифр)
```

С другой стороны, если сумма меньше $1000, то несколько полей должны были бы остаться незаполненными. Например, запись

```
   99.87
--------
12345678
```

содержит три пустых поля. Если чек будет напечатан с пустыми полями, то мошенник достаточно просто может дописать сумму чека. Для предотвращения этого большая часть систем выдачи чеков печатает в пустых полях символы звездочек:

```
***99.87
--------
12345678
```

Напишите программу, которая вводит долларовую сумму и печатает ее на чеке, вставляя, при необходимости, символы звездочек.

8.38. (*Словесный эквивалент суммы чека*) Продолжая обсуждение предыдущего примера, мы еще раз подчеркнем важность проектирования систем выдачи чеков, защищенных от подделок посредством дописывания суммы. В одном из распространенных методов защиты используется запись суммы в двух представлениях: в виде обычной цифровой записи и в виде ее словесного эквивалента. Если кто-нибудь и способен изменить числовую часть чека, то изменить величину суммы, записанную словами, чрезвычайно трудно.

Многие автоматизированные системы выдачи чека не печатают словесный эквивалент суммы. Возможно, основная причина этого упущения состоит в том, что большинство языков высокого уровня, используемых в коммерческих прикладных программах, не содержат адекватных возможностей обработки строк. Другой причиной является то, что логика записи словесных эквивалентов сумм чеков вызывает определенные трудности.

Напишите программу на C, которая вводит числовую сумму чека и выводит ее словесный эквивалент. Например, сумма *112.43* должна быть записана как

```
ONE HUNDRED TWELVE and 43/100
```

8.39. (*Код Морзе*) Возможно, самой знаменитой схемой кодирования является код (азбука) Морзе, разработанный Самуэлем Морзе в 1832 году для телеграфа. Код Морзе присваивает последовательность точек и тире каждому символу алфавита, каждой цифре и нескольким специальным символам (точке, запятой, двоеточию и точке с запятой). В звуковых системах точка представляется в виде короткого звука, а тире в виде долгого звука. Другие представления точек и тире используются со световыми системами и в системах с сигнальными флажками.

Разделяющий слова символ обозначается пробелом или, попросту говоря, отсутствием точки или тире. В звуковых системах пробел обозначается коротким временным отрезком, в течение которого звук не передается. Международная версия кода Морзе приведена на рис. 8.39.

Напишите программу, которая читает фразу на английском языке и кодирует ее, используя азбуку Морзе. Также напишите программу, которая читает фразу, закодированную азбукой Морзе, и преобразует ее в английский языковый эквивалент. Используйте один пробел между символами азбуки Морзе и три пробела между словами, записанными в этом коде.

Символ	Код	Символ	Код
A	.-	T	-
B	-...	U	..-
C	-.-.	V	...-
D	-..	W	.--
E	.	X	-..-
F	..-.	Y	-.--
G	--.	Z	--..
H	....		
I	..	**Цифры**	
J	.---	1	.----
K	-.-	2	..---
L	.-..	3	...--
M	--	4	....-
N	-.	5	.....
O	---	6	-....
P	.--.	7	--...
Q	--.-	8	---..
R	.-.	9	----.
S	...	0	-----

Рис. 8.39. Буквы алфавита, записанные в международном коде Морзе

8.40. (*Программа преобразования систем измерений*) Напишите программу, которая поможет пользователю с преобразованиями систем измерений. Ваша программа должна позволять пользователю определять в виде строк название единиц (сантиметров, литров, грамм и т.д. для метрической системы единиц и дюймов, кварт, фунтов и т.д. для английской системы измерений) и должна отвечать на простые вопросы типа:

```
"How many inches are in 2 meters?"
(Сколько дюймов в 2 метрах?)

"How many liters are in 10 quarts?"
(Сколько литров в 10 квартах?)
```

Ваша программа должна распознавать недопустимые преобразования. Например, вопрос

```
"How many feet in 5 kilograms?"
(Сколько футов в 5 килограммах?)
```

не имеет смысла, поскольку «футы» — единицы длины, а «килограмм» — единица веса.

8.41. (*Письма должникам*) Множество предпринимателей тратят время и деньги, собирая просроченные долги. Обработка должников — процесс создания повторных и настойчивых запросов должнику в попытках получить долг.

Для автоматического генерирования писем с серьезностью требований, зависящей от срока долга, часто используются компьютеры. Теория состоит в том, что, поскольку долг становится все более старым, его возврат становится все более трудным и, следовательно, такие письма должны иметь все более угрожающий характер.

Напишите программу С, которая содержит тексты пяти писем нарастающей серьезности. Ваша программа должна вводить:

1. Имя должника
2. Адрес должника
3. Счет должника
4. Сумма долга
5. Возраст долга (один просроченный месяц, два и т.д.).

Используйте возраст долга для выбора одного из пяти текстов сообщения и выводите на печать письма, вставляя в него информацию, предоставленную пользователем.

Сложный проект по теме обработки строк

8.42. (*Генератор кроссвордов*) Большинство людей наверняка сталкивались с решением кроссвордов, но немногие пытались сами составить хотя бы один. Производство кроссвордов является сложной задачей. Она предлагается здесь как проект по теме обработки строк, требующий больших усилий. В этом проекте имеется множество частных задач, которые должен решить программист, чтобы запустить в работу даже простейшую программу генерирования кроссвордов. Например, как представить сетку кроссворда внутри компьютера? Нужно ли использовать ряд строк или двумерные массивы? Программисту нужен источник слов (т.е. компьютерный словарь), к которому должна обращаться программа. В какой форме нужно хранить слова, чтобы упростить сложную обработку, требующуюся программе? По-настоящему честолюбивый читатель захочет генерировать и ту часть головоломки, в которой приводятся вопросы кроссворда. Но даже генерация пустого кроссворда является непростой задачей.



Форматированный ввод/вывод



Цели

- Понять принципы организации входных и выходных потоков.
- Научиться использовать все возможности форматирования при выводе.
- Научиться использовать все возможности форматирования при вводе.

Содержание

9.1. Введение
9.2. Потоки
9.3. Форматированный вывод с применением printf
9.4. Печать целых чисел
9.5. Печать чисел с плавающей точкой
9.6. Печать строк и символов
9.7. Другие спецификаторы преобразования
9.8. Печать с заданием ширины поля и точности представления
9.9. Использование флагов в строке управления форматом printf
9.10. Печать литералов и Esc-последовательностей
9.11. Форматированный ввод с применением scanf

Резюме • Распространенные ошибки программирования • Хороший стиль программирования • Советы по переносимости программ • Упражнения для самоконтроля • Ответы на упражнения для самоконтроля • Упражнения

9.1. Введение

Важная составная часть решения любой проблемы — представление результатов. В этой главе мы детально обсудим форматирующие возможности функций **printf** и **scanf**. Эти функции соответственно выводят данные в *стандартный поток вывода* и вводят данные из *стандартного потока ввода*. Четыре других функции, которые действуют на стандартных потоках ввода/вывода — **gets**, **puts**, **getchar** и **putchar** — обсуждались в главе 8. Включайте заголовочный файл **<stdio.h>** в программы, которые вызывают эти функции.

Часть возможностей **printf** и **scanf** обсуждалась в предыдущих главах. В этой главе суммирована вся эта информация, а также представлены многие другие аспекты форматирования. В главе 11 обсуждаются несколько других функций, включенных в стандартную библиотеку ввода/вывода (**stdio**).

9.2. Потоки

Весь ввод и вывод выполняется посредством *потоков* — последовательностей символов с построчной организацией. Каждая строка содержит нулевое или большее число символов и заканчивается символом новой строки. Стандарт языка С объявляет, что реализация ANSI С должна поддерживать строки по меньшей мере из 254 символов, включая ограничивающий символ новой строки.

При запуске программы к ней автоматически присоединяются три потока. Стандартный поток ввода обычно присоединяется к клавиатуре, а стандартный поток вывода — к устройству вывода информации на экран монитора (экрану). Операционные системы нередко позволяют *переадресовать* эти потоки на другие устройства. Третий поток — *стандартный поток ошибок* — также присоединяется к экрану. В него выводятся сообщения об ошибках. В главе 11, «Работа с файлами», потоки рассмотрены более подробно.

9.3. Форматированный вывод с применением printf

С помощью функции **printf** можно точнейшим образом форматировать вывод программы. Каждый вызов **printf** содержит *строку управления форматом*, в которой описывается формат вывода. Строка управления форматом содержит *спецификаторы преобразования*, *флаги*, *ширину полей*, *точность представления* и *литеральные символы*. Вместе с символами процента (%) они образуют *спецификации преобразования*. Функция **printf** может выполнять следующие виды форматирования, каждый из которых обсуждается в этой главе:

1. *Округление* значений с плавающей точкой до указанного числа десятичных знаков.
2. *Выравнивание* столбца чисел по положению десятичной точки в столбце.
3. *Выравнивание* выводимых данных *по правому краю* и *по левому краю*.
4. *Вставка литеральных символов* в заданное место выводимой строки.
5. Представление числа с плавающей точкой в экспоненциальном формате.
6. Представление целого без знака в восьмеричном и шестнадцатеричном формате. См. приложение Д, «Системы счисления», где приведена детальная информация по восьмеричным и шестнадцатеричным числам.
7. Вывод данных всех типов с фиксированной шириной поля и точностью представления.

Функция **printf** имеет следующую форму:

```
printf (строка_управления_форматом, другие_аргументы);
```

В *строке_управления_форматом* описывается формат вывода, а каждый из *других_аргументов* (которые не являются обязательными) сопоставляется с соответствующей спецификацией преобразования, указанной для него в

строке_управления_форматом. Каждая спецификация преобразования начинается с символа процента и заканчивается спецификатором преобразования. В одной строке управления форматом может быть указано несколько спецификаций преобразования.

Распространенная ошибка программирования 9.1

Строка_управления_форматом не заключена в кавычки.

Хороший стиль программирования 9.1

Аккуратно форматируйте представление данных при выводе. Это делает результаты программы понятнее и уменьшает число ошибок пользователя.

9.4. Печать целых чисел

Целое число — это число вида 776 или -52, т.е. не содержащее десятичной точки. Целые значения выводятся в одном из нескольких возможных форматов. На рис. 9.1 показаны все спецификаторы преобразования целых чисел.

Спецификатор	Описание
`d`	Выводит десятичное целое со знаком.
`i`	Выводит десятичное целое со знаком (Примечание: различие **i** и **d** проявляется в функции ***scanf***).
`o`	Выводит восьмеричное целое без знака.
`u`	Выводит десятичное целое без знака.
`x` или `X`	Выводит шестнадцатеричное целое без знака. Спецификатор **X** используется для вывода числа цифрами **0-9** и буквами **A-F**, а **x** — для вывода числа цифрами **0-9** и буквами **a-f**.
`h` или `l` (буква `l`)	Помещается перед спецификатором преобразования целого, чтобы показать, что выводится целое соответственно типа **short** или **long**.

Рис. 9.1. Спецификаторы преобразования целых значений

Программа, приведенная на рис. 9.2, печатает целое число, последовательно используя все спецификаторы преобразования целых. Обратите внимание, что знак выводится только для отрицательных чисел, а вывод знаков «плюс» подавляется. Позже в этой главе мы увидим, как написать формат преобразования так, чтобы знаки «плюс» также печатались. Также обратите внимание, что значение **-455** печатается с использованием формата **%u**, и на компьютере с двухбайтовыми целыми оно преобразуется в значение без знака, равное **65081**.

Распространенная ошибка программирования 9.2

Печать отрицательного значения со спецификатором, который предполагает наличие значения без знака.

```
/* Использование спецификаторов преобразования целых значений */
#include <stdio.h>

main()
{
   printf("%d\n", 455);
   printf("%i\n", 455); /* в printf i то же, что и d*/
   printf("%d\n", +455);
   printf("%d\n", -455);
   printf("%hd\n", 32000);
   printf("%ld\n", 2000000000);
   printf("%o\n", 455);
   printf("%u\n", 455);
   printf("%u\n", -455);
   printf("%x\n", 455);
   printf("%X\n", 455);
   return 0;
}
```

```
455
455
455
-455
32000
2000000000
707
455
65081
1c7
1C7
```

Рис. 9.2. Использование спецификаторов преобразования целых значений

9.5. Печать чисел с плавающей точкой

Число с плавающей точкой содержит десятичную точку, например, **33.5** или **657.983**. Числа с плавающей точкой выводятся в одном из нескольких возможных форматов, спецификаторы преобразования которых приведены на рис. 9.3.

Спецификаторы преобразования **e** и **E** выводят значение с плавающей точкой в *экспоненциальной нотации*. Экспоненциальная нотация — это компьютерный эквивалент *научной нотации*, используемой в физике. Например, в научной нотации значение 150.4582 представляется как

1.504582×10^2

а в экспоненциальной нотации компьютера

```
1.504582E+02
```

Эта нотация показывает, что **1.504582** умножается на число **10**, возведенное во вторую степень **(E+02)**. **E** — это условное обозначение «экспоненты».

Значения, которые выводятся с использованием спецификаторов преобразования **e**, **E**, и **f**, выводятся по умолчанию с точностью 6 разрядов после десятичной точки; другие значения точности могут быть заданы явным образом. Спецификатор преобразования **f** всегда выводит по меньшей мере одну цифру

слева от десятичной точки. Спецификаторы преобразования **e** и **E** печатают соответственно символ **e** нижнего регистра и символ **E** верхнего регистра, после которого следует значение экспоненты, и всегда печатают точно одну цифру слева от десятичной точки.

Спецификатор преобразования **g** (**G**) выводит число в формате **e** (**E**) или **f** без нулей справа (т.е. **1.234000** выводится как **1.234**). В формате **e** (**E**) значение числа выводится в том случае, если после его преобразования в экспоненциальную нотацию значение экспоненты оказывается меньше **-4**, либо когда оно превышает или равно заданной точности представления (6 значащих цифр по умолчанию для **g** и **G**). В других случаях для вывода значения используются спецификатор преобразования **f**. Со спецификаторами **g** или **G** нули справа в дробной части выводимого значения не печатаются. Для вывода десятичной точки требуется по меньшей мере одна значащая цифра в дробной части. Со спецификацией преобразования **%g** значения **0.0000875**, **8750000.0**, **8.75**, **87.50** и **875** печатаются как **8.75e-05**, **8.75e+06**, **8.75**, **87.5** и **875**. Значение **0.0000875** использует нотацию **e**, поскольку после преобразования в экспоненциальный формат значение экспоненты оказывается меньше **-4**. Для вывода значения **8750000.0** также используется нотация **e**, так как значение экспоненты равно точности представления, задаваемой по умолчанию.

Спецификатор	Назначение
`e` или `E`	Выводит значение с плавающей точкой в экспоненциальной нотации.
`f`	Выводит значение с плавающей точкой.
`g` или `G`	Выводит значение с плавающей точкой либо в формате **f**, либо в экспоненциальном формате **e** (или **E**).
`L`	Помещается перед любым спецификатором преобразования значения с плавающей точкой, чтобы показать, что в данном случае выводится значение типа **long double**.

Рис. 9.3. Спецификаторы преобразования значений с плавающей точкой

Точность представления для спецификаторов преобразования **g** и **G** показывает максимальное число печатающихся значащих цифр, включая цифру слева от десятичной точки. С использованием спецификации **%g** значение **1234567.0** будет напечатано как **1.23457e+06** (помните, что все спецификаторы преобразования значений с плавающей точкой имеют по умолчанию точность представления, равную 6). Обратите внимание, что выведенный результат содержит 6 значащих цифр. Когда значение выводится в экспоненциальной нотации, разница между **g** и **G** аналогична разнице между **e** и **E**: спецификатор **g** выводит символ нижнего регистра **e**, а спецификатор **G** выводит символ верхнего регистра **E**.

Хороший стиль программирования 9.2

При выводе данных позаботьтесь о том, чтобы пользователь отдавал себе отчет о ситуациях, в которых форматирование может давать неточные данные (например, ошибки округления из-за ограниченного значения точности).

Программа, приведенная на рис. 9.4, демонстрирует три вида спецификаций преобразования значений с плавающей точкой. Обратите внимание, что спецификации преобразования **%E** и **%g** выводят округленные значения.

```
/* Печать чисел с плавающей точкой с использованием
   спецификаторов преобразования значений с плавающей точкой */
#include <stdio.h>

main()
{
   printf("%e\n", 1234567.89);
   printf("%e\n", +1234567.89);
   printf("%e\n", -1234567.89);
   printf("%E\n", 1234567.89);
   printf("%f\n", 1234567.89);
   printf("%g\n", 1234567.89);
   printf("%G\n", 1234567.89);

   return 0;
}
```

```
1.234568e+06
1.234568e+06
-1.234568e+06
1.234568E+06
1234567.890000
1.23457e+06
1.23457E+06
```

Рис. 9.4. Использование спецификаторов преобразования значений с плавающей точкой

9.6. Печать строк и символов

Для вывода отдельных символов и строк используются соответственно спецификаторы преобразования **c** и **s**. Для спецификатора преобразования **c** требуется аргумент типа **char**. Для спецификатора преобразования **s** в качестве аргумента используется указатель на **char**. Спецификатор преобразования **s** выводит символы до тех пор, пока не встретится ограничительный символ **NULL** (**'\0'**). Программа, показанная на рис. 9.5, выводит символы и строки со спецификаторами преобразования **c** и **s**.

Распространенная ошибка программирования 9.3

Использование **%c** для вывода первого символа строки. Спецификация преобразования **%c** предполагает аргумент типа **char**. Строка — это указатель на **char**, т.е. **char ***.

Распространенная ошибка программирования 9.4

Использование **%s** для вывода аргумента типа **char**. Спецификация преобразования **%s** предполагает передачу в качестве аргумента указателя на **char**. На некоторых системах это приведет к фатальной ошибке при выполнении программы, называемой «несанкционированным доступом».

Распространенная ошибка программирования 9.5

Использование одинарных кавычек для обозначения строки символов есть синтаксическая ошибка. Строка символов должна быть заключена в двойные кавычки.

Распространенная ошибка программирования 9.6

Использование двойных кавычек для обозначения символьной константы. Такая запись фактически создает строку, содержащую два символа, вторым из которых является символ **NULL**, ограничивающий строку. Символьная константа содержит один символ и должна заключаться в одинарные кавычки.

```
/* Печать строк и символов */
#include <stdio.h>

main()
{
   char character = 'A';
   char string [] = "This is a string";
   char *stringPtr = "This is also a string";

   printf("%c\n", character);
   printf("%s\n", "This is a string");
   printf("%s\n", string);
   printf("%s\n", stringPtr);
   return 0;
}
```

```
A
This is a string
This is a string
This is also a string
```

Рис. 9.5. Использование спецификаторов преобразования строк и символов

9.7. Другие спецификаторы преобразования

Тремя оставшимися спецификаторами преобразования являются спецификаторы **p**, **n** и **%** (рис. 9.6).

Совет по переносимости программ 9.1

Спецификатор преобразования **p** выводит адрес, на который ссылается указатель, в зависящем от конкретной реализации виде (на многих системах вместо десятичной нотации используется шестнадцатеричная).

Спецификатор преобразования **n** сохраняет количество символов, уже выведенных текущим оператором **printf**, а соответствующий аргумент является указателем на целую переменную, в которую помещается это значение. Спецификация преобразования **%n** ничего не печатает. Спецификатор преобразования **%** просто выводит знак процента.

Спецификатор	Описание
p	Выводит значение указателя в форме, зависящей от конкретной реализации.
n	Сохраняет число символов, уже выведенных текущим оператором **printf**. Соответствующий аргумент является указателем на целое значение. Ничего не выводит.
%	Выводит символ процента.

Рис. 9.6. Другие спецификаторы преобразования

В программе, приведенной на рис. 9.7, спецификация **%p** печатает значение **ptr** и адреса **x**; эти значения идентичны, поскольку **ptr** присваивается адрес **x**. Далее **%n** сохраняет число выведенных третьим оператором **printf** символов в целой переменной **y**, после чего печатается значение **y**. Последний оператор **printf** использует спецификацию **%%**, чтобы напечатать символ **%** в строке символов. Обратите внимание, что каждый вызов **printf** возвращает или число выведенных символов, или отрицательное значение, если при выводе произошла ошибка.

```
/* Применение спецификаторов преобразования p, n, и % */
#include <stdio.h>

main()
{
   int *ptr;
   int x = 12345, y;

   ptr = &x;
   printf("The value of ptr is %p\n", ptr);
   printf("The address of x is %p\n\n", &x);

   printf("Total characters printed on this line is:%n", &y);
   printf(" %d\n\n", y);

   y = printf("This line has 28 characters\n");
   printf("%d characters were printed\n\n", y);

   printf("Printing a %% in a format control string\n");
   return 0;
}
```

```
The value of ptr is 001F2BB4
The address of x is 001F2BB4

Total characters printed on this line is: 41

This line has 28 characters
8 characters were printed

Printing a % in a format control string
```

Рис. 9.7. Применение спецификаторов преобразования **p**, **n** и **%**

Распространенная ошибка программирования 9.7

Попытка напечатать символ процента как литерал, используя в строке управления форматом **%**, а не **%%**. Если в строке управления форматом появляется символ **%**, то за ним должен следовать спецификатор преобразования.

9.8. Печать с заданием ширины поля и точности представления

Точный размер поля, в котором печатаются данные, задается *шириной поля*. Если ширина поля больше, чем необходимо для печати данных, то данные обычно выравниваются внутри поля по его правому краю. Целое число, задающее ширину поля, вставляется в спецификацию преобразования между знаком процента (%) и спецификатором преобразования. Программа, приведенная на рис. 9.8, печатает две группы из пяти чисел, выравнивая по правому краю те из них, которые содержат меньшее количество цифр, чем задано шириной поля. Обратите внимание, что для вывода значений, превышающих текущее значение ширины поля, она автоматически увеличивается, и что знак «минус» отрицательного значения занимает одну символьную позицию поля. Ширина поля может использоваться со всеми спецификаторами преобразования.

```
/* Вывод целых чисел с выравниванием по правому краю */
#include <stdio.h>

main()
{
   printf("%4d\n", 1);
   printf("%4d\n", 12);
   printf("%4d\n", 123);
   printf("%4d\n", 1234);
   printf("%4d\n\n", 12345);

   printf("%4d\n", -1);
   printf("%4d\n", -12);
   printf("%4d\n", -123);
   printf("%4d\n", -1234);
   printf("%4d\n", -12345);

   return 0;
}
```

```
   1
  12
 123
1234
12345

  -1
 -12
-123
-1234
-12345
```

Рис. 9.8. Выравнивание целых чисел по правому краю поля

Распространенная ошибка программирования 9.8

Задание недостаточной ширины поля для вывода текущего значения. Это может сместить другие данные, спутав все результаты. Следите за своими данными!

Функция **printf** дает также возможность задать *точность представления*, с которой будут напечатаны данные. Точность имеет различный смысл для различных типов данных. Если она используется со спецификаторами преобразования целых чисел, то показывает минимальное количество цифр, которое должно быть выведено. Если выводимое значение содержит меньше цифр, чем задано точностью, то перед ним будут напечатаны префиксные нули, так чтобы общее количество цифр стало равно заданной точности. Для целых чисел точность по умолчанию равна 1. Если точность используется со спецификаторами преобразования значений с плавающей точкой **e**, **E** и **f**, то точность — это количество цифр, которое будет напечатано после десятичной точки. Для спецификаторов преобразования **g** и **G** точность — это максимальное количество значащих цифр, которое будет выведено. Для спецификатора преобразования **s** точность — это максимальное число символов строки, которое будет напечатано. Чтобы использовать точность представления, поместите между знаком процента и спецификатором преобразования десятичную точку (.) с последующим целым числом, задающим точность представления данных. Программа, приведенная на рис. 9.9, показывает варианты задания точности представления данных в строках управления форматом. Обратите внимание, что если при печати значений с плавающей точкой задана меньшая точность, чем число десятичных разрядов дробной части исходного значения, то это значение округляется.

Ширина поля и точность представления могут быть объединены, для чего между знаком процента и спецификатором преобразования нужно вставить значение ширины поля, десятичную точку и последующее значение точности, как в следующем операторе:

```
printf("%9.3f", 123.456789);
```

Он выводит **123.457** с тремя цифрами справа от десятичной точки и выравнивает результат по правому краю поля, шириной 9 цифр.

```
/* Использование точности представления при печати целых чисел,
   чисел с плавающей точкой и строк */
#include <stdio.h>

main()
{
   int i = 873;
   float f = 123.94536;
   char s[] = "Happy Birthday";

   printf("Using precision for integers\n");
   printf("\t%.4d\n\t%.9d\n\n", i, i);
   printf("Using precision for floating-point numbers\n");
   printf("\t%.3f\n\t%.3e\n\t%.3g\n\n", f, f, f);
   printf("Using precision for strings\n");
   printf("\t%.11s\n", s);
   return 0;
}
```

Рис. 9.9. Использование точности представления для вывода информации различного типа (часть 1 из 2)

```
Using precision for integers
        0873
        000000873

Using precision for floating-point numbers
        123.845
        1.239e+02
        124

Using precision for strings
        Happy Birth
```

Рис. 9.9. Использование точности представления для вывода информации различного типа (часть 2 из 2)

Ширину поля и точность представления можно задать, используя целочисленные выражения в списке аргументов после строки управления форматом. Чтобы это сделать, вставьте * (звездочку) вместо ширины поля или точности (или вместо того и другого). Звездочки при печати будут заменены соответствующими значениями из списка аргументов. Значение аргумента для ширины поля может быть отрицательным, но для точности представления должно быть только положительным. Отрицательное значение ширины поля приводит к выравниванию вывода по левому краю поля, как описано в следующем разделе. Оператор

```
printf("%*.*f", 7, 2, 98.736);
```

использует значение **7** для ширины поля, **2** для точности представления и выводит значение **98.74** с выравниванием по правому краю поля.

9.9. Использование флагов в строке управления форматом printf

Функция **printf** предусматривает также использование *флагов*, дополняющих возможности форматирования. Пользователю доступны пять флагов строки управления форматом (рис. 9.10).

Чтобы использовать в спецификации преобразования флаг, поместите его непосредственно справа от знака процента. В одной спецификации могут быть объединены несколько флагов.

Программа, приведенная на рис. 9.11, демонстрирует выравнивание строки, целого числа, символов и числа с плавающей точкой по правому и по левому краю поля.

Флаг	Описание
- (знак "минус")	Выравнивание вывода по левому краю в пределах заданной ширины поля.
+ (знак "плюс")	Вывод знака плюс перед положительными значениями и знака минус перед отрицательными.
пробел	Вывод пробела перед положительным значением при отсутствии в спецификации преобразования флага +.

Флаг	Описание
#	Печать префикса **0** перед выводимым значением при использовании спецификатора преобразования восьмеричных значений **o**.
	Печать префикса **0x** или **0X** перед выводимым значением при использовании спецификатора преобразования шестнадцатеричных значений **x** или **X**.
	При использовании со спецификаторами преобразования **e**, **E**, **f**, **g** или **G** вынуждает печатать десятичную точку для чисел с плавающей точкой, которые не содержат дробной части. (Обычно десятичная точка выводится только при наличии хотя бы *одной цифры справа от нее.*) *Для спецификаторов* **g** и **G** отменяет подавление правых нулей дробной части.
0 (нуль)	Дополняет поле до заданной ширины нулями слева.

Рис. 9.10. Флаги строки управления форматом

Программа рис. 9.12 печатает положительное и отрицательное число при включении в спецификацию флага преобразования + и без него. Обратите внимание, что знак «минус» выводится в обоих случаях, но знак «плюс» отображается только при указании флага +.

```
/* Выравнивание по правому и по левому краю поля */
 #include <stdio.h>

main()
{
   printf("%10s%10d%10c%10f\n\n", "hello", 7, 'a', 1.23);
   printf("%-10s%-10d%-10c%-10f\n", "hello", 7, 'a', 1.23);
   return 0;
}
```

```
     hello         7         a  1.230000
hello     7         a         1.230000
```

Рис. 9.11. Выравнивание строк по правому и по левому краю поля

```
/* Печать чисел с флагом + и без него */
#include <stdio.h>

main()
{
   printf("%d\n%d\n", 786, -786);
   printf("%+d\n%+d\n", 786, -786);
   return 0;
}
```

```
786
-786
+786
-786
```

Рис. 9.12. Печать положительных и отрицательных чисел с флагом + и без него

Программа рис. 9.13 выводит префиксные пробелы с положительным числом при включении в спецификацию преобразования флага пробела. Это полезно для выравнивания положительных и отрицательных чисел с одинаковым числом цифр.

```
/* Вывод пробелов перед значениями со знаком, которые
   не сопровождаются + или - */
#include <stdio.h>

main()
{
   printf("% d\n% d\n", 547, -547);
   return 0;
}
```

```
 547
-547
```

Рис. 9.13. Использование флага пробела

Программа, приведенная на рис. 9.14, использует флаг #, чтобы вывести префикс **0** для восьмеричного значения, **0x** и **0X** для шестнадцатеричных значений и использует принудительный вывод десятичной точки для значения, печатающегося со спецификатором **g**.

```
/* Использование флага # со спецификаторами преобразования o, x,
X
   и любым спецификатором значений с плавающей точкой */
#include <stdio.h>

main()
{
   int c = 1427;
   float p = 1427.0;

   printf("%#o\n", c);
   printf("%#x\n", c);
   printf("%#X\n", c);
   printf("\n%g\n", p);
   printf("%#g\n", p);
   return 0;
}
```

```
02623
0x593
0X593

1427
1427.00
```

Рис. 9.14. Применение флага #

Программа, приведенная на рис. 9.15, использует комбинацию флагов **+** и **0** (нуль) для вывода **452** в поле шириной **9** позиций, со знаком **+** и заполняющими нулями. После этого программа выводит значение **452** еще раз, используя только флаг **0** и поле шириной **9** позиций.

```
/* Вывод с флагом 0(нуль) */
#include <stdio.h>

main()
{
   printf("%+09d", 452);
   printf("%09d", 452);
   return 0;
}
```

```
+00000452
000000452
```

Рис. 9.15. Использование флага **0** (нуль)

9.10. Печать литералов и Esc-последовательностей

Большинство литералов, которые выводятся оператором **printf**, могут быть просто включены в строку управления форматом. Однако существуют отдельные «проблемные» символы типа кавычек ("), которые ограничивают саму строку управления форматом. Различные управляющие символы типа символа новой строки и символа табуляции должны быть представлены соответствующей *escape-последовательностью*, или *escape-кодом*. Esc-последовательность представляет собой обратную косую черту (\) с последующим *escape-символом*. В таблице на рис. 9.16 приведен список всех Esc-последовательностей и действий, которые они вызывают.

Esc-последовательность	Описание
\'	Вывод символа одинарной кавычки (').
\"	Вывод символа двойной кавычки(").
\?	Вывод символа вопросительного знака (?).
\\	Вывод символа обратной косой черты (\ ).
\a	Вызывает звуковой сигнал (звонок) или визуальное предупреждение.
\b	*Перемещает курсор на одну позицию назад в текущей строке.*
\f	Перемещает курсор на начало следующей логической страницы.
\n	Перемещает курсор на начало следующей строки.
\r	Перемещает курсор на начало текущей строки.
\t	Перемещает курсор на следующую позицию горизонтальной *табуляции*.
\v	Перемещает курсор на следующую позицию вертикальной табуляции.

Рис. 9.16. Esc-последовательности

Распространенная ошибка программирования 9.9

Попытка вывести в операторе **printf** в качестве литеральных данных символ двойной кавычки, вопросительного знака или обратной косой черты, не предваряя этот символ обратной косой чертой.

9.11. Форматированный ввод с применением scanf

Ввод с точным контролем формата выполняется функцией **scanf**. Каждый оператор **scanf** содержит строку управления форматом, которая описывает формат входных данных. Строка управления форматом состоит из спецификаций преобразования и литеральных символов. Функция **scanf** имеет следующие возможности форматирования входных данных:

1. Ввод всех типов данных.
2. Ввод определенных символов из входного потока.
3. Пропуск определенных символов из входного потока.

Функция **scanf** имеет следующий формат:

```
scanf(строка_управления_форматом, другие_аргументы);
```

В *строке_управления_форматом* содержится описание входного формата, а *другие_аргументы* — это указатели на переменные, в которых сохраняются введенные данные.

Хороший стиль программирования 9.3

При вводе данных просите пользователя ввести один элемент или небольшое число элементов данных за один раз. Не предлагайте ему вводить в ответ на один запрос значительный объем данных.

На рис. 9.17 приведены спецификаторы преобразования, используемые при вводе всех типов данных. В оставшейся части этого раздела приведены программы, которые демонстрируют чтение данных с различными спецификаторами преобразования функции **scanf**.

Спецификатор	Описание
Целые	
`d`	Прочитать десятичное целое число с необязательным знаком. Соответствующий аргумент – указатель на целое число.
`i`	Прочитать десятичное, восьмеричное или шестнадцатеричное целое число с необязательным знаком. Соответствующий аргумент – указатель на целое число.
`o`	Прочитать восьмеричное число. Соответствующий аргумент – указатель на целое число без знака.
`u`	Прочитать десятичное целое число без знака. Соответствующий аргумент – указатель на целое число без знака.
`x` или `X`	Прочитать шестнадцатеричное число. Соответствующий аргумент – указатель на целое число без знака.
`h` или `l`	Помещается перед любым спецификатором преобразования целых, чтобы показать, что вводится целое соответственно типа **short** или **long**.
Числа с плавающей точкой	
`e, E, f, g` или `G`	Прочитать значение с плавающей точкой. Соответствующий аргумент – указатель на переменную с плавающей точкой.

Спецификатор	Описание
l или L	Помещается перед любым спецификатором преобразования значений с плавающей точкой, чтобы показать, что вводится значение типа **double** или **long double**.
Символы и строки	
c	Прочитать символ. Соответствующий аргумент – указатель на **char**. Нулевой символ (**'\0'**) не добавляется.
s	Прочитать строку. Соответствующий аргумент – указатель на массив типа **char**. Массив должен иметь достаточный размер для хранения строки и ограничивающего нулевого символа (**'\0'**).
Набор сканирования	
[*символы*]	Сканирование входного потока и ввод символов, входящих в набор, с сохранением введенных данных в массиве.
Разное	
p	Прочитать адрес, в том же формате, который генерируется при выводе адреса со спецификатором **%p** в операторе **printf**.
n	Сохранить число символов, введенных до настоящего момента данным оператором **scanf**. Соответствующий аргумент – указатель на целое.
%	Пропустить при вводе знак процента (**%**).

Рис. 9.17. Спецификаторы преобразования для **scanf**

Программа, приведенная на рис. 9.18, читает целые числа с различными спецификаторами преобразования целых чисел и выводит их в десятичной нотации. Обратите внимание, что спецификатор **%i** дает возможность вводить десятичные, восьмеричные и шестнадцатеричные целые числа.

```
/* Чтение целых чисел */
#include <stdio.h>

main()
{
   int a, b, c, d, e, f, g;

   printf("Enter seven integers: ");
   scanf("%d%i%i%i%o%u%x", &a, &b, &c, &d, &e, &f, &g);
   printf("The input displayed as decimal integers is:\n");
   printf("%d %d %d %d %d %d %d\n", a, b, c, d, e, f, g);
   return 0;
}
```

```
Enter seven integers: -70 -70 070 0x70 70 70 70
The input displayed as decimal integers is:
-70 -70 56 112 56 70 112
```

Рис. 9.18. Чтение входных данных со спецификаторами преобразования целых чисел

При вводе чисел с плавающей точкой может быть использован любой из спецификаторов преобразования значений с плавающей точкой: **e**, **E**, **f**, **g** или **G**. Программа на рис. 9.19 читает три числа с плавающей точкой с тремя видами спецификаторов преобразования и выводит их со спецификатором преобразования **f**. Обратите внимание, что результат исполнения программы подтверждает тот факт, что выведенные значения с плавающей точкой не являются точными. Об этом свидетельствует второе по счету напечатанное значение.

```
/* Чтение значений с плавающей точкой */
#include <stdio.h>

main()
{
   float a, b, c;

   printf("Enter three floating-point numbers: \n");
   scanf("%e%f%g", &a, &b, &c);
   printf("Here are the numbers entered in plain\n");
   printf("floating-point notation:\n");
   printf("%f %f %f\n", a, b, c);
   return 0;
}
```

```
Enter three floating-point numbers:
1.27987 1.27987e+03 3.38476e-06
Here are the numbers entered in plain
floating-point notation
1.279870
1279.869995
0.000003
```

Рис. 9.19. Чтение входных данных со спецификаторами преобразования значений с плавающей точкой

Символы и строки вводятся с использованием спецификаторов преобразования соответственно **c** и **s**. Программа на рис. 9.20 предлагает пользователю ввести строку. Программа считывает первый символ строки со спецификацией **%c** и сохраняет его в символьной переменной **x**; затем вводит оставшуюся часть строки со спецификацией **%s** и сохраняет ее в массиве символов **y**.

Последовательность символов можно ввести, используя *набор сканирования*. Набор сканирования — это список символов, заключенный в квадратные скобки **[]**, следующий сразу за знаком процента в строке управления форматом. Набор сканирования просматривает символы во входном потоке, выбирая только те из них, которые содержатся в наборе символов. Каждый раз при совпадении символа он сохраняется в соответствующем аргументе набора сканирования. Аргумент — это указатель на массив символов. Набор сканирования прекращает читать символы после того, как встречается символ, не содержащийся в наборе. Если уже первый символ входного потока не соответствует ни одному из символов набора, то в массиве сохраняется только нулевой символ. Программа на рис. 9.21 использует набор символов сканирования **[aeiou]** для отбора гласных из входного потока. Обратите внимание, что считываются только первые семь введенных символов. Восьмой символ (**h**) не содержится в заданном наборе и ввод завершается.

```
/* Чтение символов и строк */
#include <stdio.h>

main()
{
   char x, y[9];

   printf("Enter a string: ");
   scanf("%c%s", &x, y);

   printf("The input was:\n");
   printf("the character \"%c\" ", x);
   printf("and the string \"%s\"\n", y);
   return 0;
}
```

```
Enter a string: Sunday
The input was:
the character "S" and the string "unday"
```

Рис. 9.20. Ввод символов и строк

```
/* Применение набора сканирования */
#include <stdio.h>

main()
{
   char z[9];

   printf("Enter a string: ");
   scanf("%[aeiou]", z);
   printf("The input was \"%s\"\n", z);
   return 0;
}
```

```
Enter a string: ooeeooahah
The input was "ooeeooa"
```

Рис. 9.21. Применение набора сканирования

Набор сканирования можно также использовать для ввода символов, не содержащихся в заданном списке, если применить *инвертированный набор сканирования*. Чтобы инвертировать набор символов, поместите символ ^ в квадратные скобки перед первым символом набора. В этом случае в массиве будут сохраняться только те символы, которые не входят в набор. Инвертированный набор завершает ввод после появления во входном потоке символа, который имеется в наборе. Программа на рис. 9.22 использует набор **[^aeiou]** для поиска во входном потоке символов, не являющихся гласными.

Для чтения определенного числа символов из входного потока со спецификацией преобразования **scanf** может использоваться значение ширины поля. Программа на рис. 9.23 читает ряд последовательных цифр как целое число с двумя десятичными разрядами и целое число, состоящее из остатка цифр входного потока.

```
/* Применение инвертированного набора сканирования */
#include <stdio.h>

main()
{
   char z[9];

   printf("Enter a string: ");
   scanf("%[^aeiou]", z);
   printf("The input was \"%s\"\n", z);
   return 0;
}
```

```
Enter a string: String
The input was "Str"
```

Рис. 9.22. Применение инвертированного набора сканирования

```
/* Ввод данных с заданием ширины поля */
#include <stdio.h>

main()
{
   int x, y;

   printf("Enter a six digit integer: ");
   scanf("%2d%d", &x, &y);
   printf("The integers input were %d and %d\n", x, y);
   return 0;
}
```

```
Enter a six digit integer: 123456
The integers input were 12 and 3456
```

Рис. 9.23. Ввод данных с заданием ширины поля

Часто бывает необходимо пропустить некоторые символы входного потока. Например, дата могла бы быть введена как

```
7-9-91
```

Каждое число даты требуется сохранить, но символы тире, которыми они разделены, могут быть отброшены. Чтобы устранить ненужные символы, включите их в строку управления форматом **scanf** (пробельные символы — пробел, новая строка и табуляция — пропускают все пробельные символы, предшествующие значимым данным). Например, чтобы пропустить символы тире при вводе даты, используйте оператор

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

Хотя **scanf** и устраняет тире в предыдущем примере, дата может быть введена и в следующем формате:

```
7/9/91
```

В этом случае предыдущий оператор **scanf** не пропустил бы ненужные символы. По этой причине **scanf** имеет возможность установить *символ подавления присваивания* ***** (звездочка). Символ подавления присваивания дает возможность **scanf** читать любой тип данных и отбрасывать их, не присваивая пе-

ременной. Программа на рис. 9.24 использует символ подавления присваивания в спецификации преобразования %c для указания того, что символ, появляющийся во входном потоке, должен быть прочтен и отброшен. Сохраняются только значения месяца, дня и года. Для демонстрации корректности ввода программа печатает введенные значения. Обратите внимание, здесь нет таких переменных в списке аргументов, которые соответствовали бы спецификациям преобразования с символом подавления присваивания, поскольку для этих спецификаций преобразования присваивание не выполняется.

```
/* Чтение и отбрасывание символов из входного потока */
#include <stdio.h>

main()
{
   int month1, day1, year1, month2, day2, year2;

   printf("Enter a date in the form mm-dd-yy: ");
   scanf("%d%*c%d%*c%d", &month1, &day1, &year1);
   printf("month = %d day = %d year = %d\n\n",
           month1, day1, year1);
   printf("Enter a date in the form mm/dd/yy: ");
   scanf("%d%*c%d%*c%d", &month2, &day2, &year2);
   printf("month = %d day = %d year = %d\n",
           month2, day2, year2);
   return 0;
}
```

```
Enter a date in the form mm-dd-yy: 11-18-71
month = 11  day = 18  year = 71

Enter a date in the form mm/dd/yy: 11/18/71
month = 11  day = 18  year = 71
```

Рис. 9.24. Чтение и отбрасывание символов из входного потока

Резюме

- Весь ввод и вывод выполняется посредством потоков — последовательностей символов с построчной организацией. Каждая строка содержит нулевое или большее число символов и заканчивается символом новой строки.
- Стандартный поток ввода обычно присоединяется к клавиатуре, а стандартный поток вывода — к экрану.
- Операционные системы нередко позволяют переадресовать стандартный входной и выходной потоки другим устройствам.
- Строка управления форматом **printf** описывает формат выводимых данных и содержит спецификаторы преобразования, флаги, ширину полей, точность представления и литеральные символы.
- Для печати целых значений используются следующие спецификаторы преобразования: **d** или **i** для целых с необязательным знаком, **o** для восьмеричных целых без знака, **u** для десятичных целых без знака и **x** или **X** для шестнадцатеричных целых без знака. Модификаторы **h** или

l, использующиеся в качестве префикса с вышеперечисленными спецификатором преобразования, указывают на то, что целое значение имеет соответственно тип **short** или **long**.

- Для печати значений с плавающей точкой используются следующие спецификаторы преобразования: **e** или **E** для экспоненциальной нотации, **f** для обычной нотации чисел с плавающей точкой и **g** или **G**, которые выводят данные либо в нотации **e** (или **E**), либо в нотации **f**. Если указан спецификатор преобразования **g** (или **G**), то в тех случаях, когда значение экспоненты меньше **-4**, либо больше или равно точности представления, с которой выводится данное значение, то при выводе используется нотация **e** (или **E**).
- Для спецификаторов преобразования **g** и **G** точность представления задает максимальное число выводимых значащих цифр.
- Спецификатор преобразования **c** используется для печати символов.
- Спецификатор преобразования **s** используется для печати строк, ограниченных нулевым символом.
- Спецификатор преобразования **p** выводит адрес, на который ссылается указатель, в формате, зависящем от конкретной реализации (на многих системах для этого используется шестнадцатеричная нотация).
- Спецификатор преобразования **n** сохраняет число символов, уже выведенных текущим оператором **printf**. Соответствующий аргумент является указателем на целое число.
- Спецификатор преобразования **%%** выводит символ процента (%).
- Если значение ширины поля больше выводимого объекта, то он обычно выравнивается в поле по его правому краю.
- Ширина поля может использоваться со всеми спецификаторами преобразования.
- Точность представления, используемая со спецификаторами преобразования целых чисел, показывает минимальное число цифр, которое должно быть выведено. Если выводимое значение содержит меньшее количество цифр, то перед ним печатаются префиксные нули, так чтобы общее количество цифр стало равно заданной точности.
- Если точность представления используется со спецификаторами преобразования значений с плавающей точкой **e**, **E** и **f**, то точность — это число цифр, которые будут напечатаны после десятичной точки.
- Для спецификаторов преобразования **g** и **G** точность — это количество выводимых значащих цифр.
- Для спецификатора преобразования **s** точность — это число символов, которое будет выведено.
- Ширина поля и точность представления могут быть объединены, для чего между знаком процента и спецификатором преобразования вставить значение ширины поля, десятичную точку и последующее значение точности.
- Ширину поля и точность представления можно задать, используя целочисленные выражения в списке аргументов после строки управления

форматом. Чтобы это сделать, вставьте * (звездочку) вместо ширины поля или точности (или вместо того и другого). Звездочки при печати будут заменены соответствующими значениями из списка аргументов. Значение аргумента для ширины поля может быть отрицательным, но для точности представления оно должно быть только положительным.

- Флаг - (минус) выравнивает выводимое значение по левому краю поля.
- Флаг + печатает знак «+» перед положительными значениями и знак «-» перед отрицательными.
- Флаг пробела печатает пробел перед положительным значением, если для его вывода не используется флаг +.
- Флаг # печатает префикс **0** перед восьмеричными значениями, префикс **0x** или **0X** перед шестнадцатеричными значениями. При использовании со спецификаторами преобразования **e**, **E**, **f**, **g** или **G** выводит десятичную точку со всеми числами с плавающей точкой (обычно десятичная точка выводится только при наличии у числа значимой дробной части).
- Флаг **0** выводит перед числом нули, если оно не занимает целиком все поле.
- Ввод данных с точным контролем формата выполняется библиотечной функцией **scanf**.
- Целые значения вводятся со спецификатором преобразования **d** и **i** для целых с необязательным знаком и спецификаторами **o**, **u**, **x** и **X** для целых без знака. Для ввода целых типа **short** и **long** используются модификаторы **h** и **l** соответственно, которые помещаются перед спецификатором преобразования целых значений.
- Значения с плавающей точкой вводятся со спецификаторами преобразования **e**, **E**, **f**, **g** или **G**. Перед любым спецификатором преобразования значений с плавающей точкой можно поместить модификатор **l** или **L** для указания того, что вводимое значение имеет тип **double** или **long double** соответственно.
- Для ввода символов используется спецификатор преобразования **c**.
- Для ввода строк используется спецификатор преобразования **s**.
- Набор сканирования просматривает символы во входном потоке, выбирая только те из них, которые содержатся в наборе. Каждый раз при совпадении символа он сохраняется в массиве символов. Набор сканирования прекращает читать символы после того, как встречается символ, не содержащийся в наборе.
- Чтобы создать инвертированный набор сканирования, поместите символ ^ в квадратные скобки перед первым символом набора. В этом случае в массиве будут сохраняться только те символы, которые не входят в набор. Инвертированный набор сканирования завершает ввод после появления во входном потоке символа, который имеется в наборе.
- Значения адреса вводятся со спецификатором преобразования **p**.
- Спецификатор преобразования **n** сохраняет число символов, уже введенных текущим оператором **scanf**. Соответствующий аргумент указывает на тип **int**.

- Спецификатор преобразования %% вводит единичный символ %.
- Для чтения и отбрасывания данных из входного потока используется символ подавления присваивания (*).
- Ширина поля используется с функцией **scanf** для чтения определенного числа символов из входного потока.

Терминология

* в качестве точности
* в качестве ширины поля
escape-код
escape-код \'
escape-код \"
escape-код \?
escape-код \\
escape-код \a
escape-код \b
escape-код \f
escape-код \n
escape-код \r
escape-код \t
escape-код \v
escape-последовательность
printf
scanf
<sdtio.h>
восьмеричный формат
вставка отображаемых символов
вставка пробелов
выравнивание
выравнивание по левому краю
выравнивание по правому краю
инвертированный набор сканирования
литеральные символы
набор сканирования
научная нотация
округление
переадресация потока
поток
поток стандартного ввода
поток стандартного вывода
поток стандартной ошибки
пробельные символы
символ ^
символ подавления присваивания (*)
спецификатор преобразования %
спецификатор преобразования **c**
спецификатор преобразования **d**
спецификатор преобразования **e** или **E**
спецификатор преобразования **f**
спецификатор преобразования **g** или **G**
спецификатор преобразования **h**
спецификатор преобразования **i**
спецификатор преобразования **L**
спецификатор преобразования **l**
спецификатор преобразования **n**
спецификатор преобразования **o**
спецификатор преобразования **p**
спецификатор преобразования **s**
спецификатор преобразования **u**
спецификатор преобразования **x** (или **X**)
спецификаторы преобразования
спецификаторы преобразования целых
спецификация преобразования
строка управления форматом
точность
флаг
флаг - (минус)
флаг-пробел
флаг #
флаг + (плюс)
флаг **0** (нуль)
формат целого без знака
формат целого со знаком
целое типа **long**
целое типа **short**
число с плавающей точкой
шестнадцатеричный формат
ширина поля
экспоненциальный формат с плавающей точкой

Распространенные ошибки программирования

9.1. Строка управления форматом не заключена в кавычки.

9.2. Печать отрицательного значения со спецификатором, который предполагает наличие значения без знака.

9.3. Использование **%c** для вывода первого символа строки. Спецификация преобразования **%c** предполагает аргумент типа **char**. Строка — это указатель на **char**, т.е. **char ***.

9.4. Использование **%s** для вывода аргумента типа **char**. Спецификация преобразования **%s** предполагает передачу в качестве аргумента указателя на **char**. На некоторых системах это приведет к фатальной ошибке при выполнении программы, называемой «несанкционированным доступом».

9.5. Использование одинарных кавычек, заключающих строки символов, есть синтаксическая ошибка. Строка символов должна быть заключена в двойные кавычки.

9.6. Использование двойных кавычек для обозначения символьной константы. Такая запись фактически создает строку, содержащую два символа, вторым из которых является символ **NULL**, ограничивающий строку. Символьная константа содержит один символ и должна заключаться в одинарные кавычки.

9.7. Попытка напечатать символ процента как литерал, используя в строке управления форматом %, а не %%. Если в строке управления форматом появляется символ %, то за ним должен следовать спецификатор преобразования.

9.8. Задание недостаточной ширины поля для вывода текущего значения. Это может сместить другие данные, спутав все результаты. Следите за своими данными!

9.9. Попытка вывести в операторе **printf** в качестве литеральных данных символ двойной кавычки, вопросительного знака или обратной косой черты, не предваряя этот символ обратной косой чертой.

Хороший стиль программирования

9.1. Аккуратно форматируйте представление данных при выводе. Это делает результаты программы понятнее и уменьшает число ошибок пользователя.

9.2. При выводе данных позаботьтесь о том, чтобы пользователь отдавал себе отчет о ситуациях, в которых форматирование может давать неточные данные (например, ошибки округления из-за ограниченного значения точности).

9.3. При вводе данных просите пользователя ввести один элемент или небольшое число элементов данных за один раз. Не предлагайте ему вводить в ответ на один запрос значительный объем данных.

Советы по переносимости программ

9.1. Спецификатор преобразования **p** выводит адрес, на который ссылается указатель, способом, в зависящем от конкретной реализации виде (на многих системах вместо десятичной нотации используется шестнадцатеричная).

Упражнения для самоконтроля

9.1. Заполните пропуски в следующих утверждениях:

a) Весь ввод и вывод организован в виде _____.

b) _____ поток _____ обычно подключен к клавиатуре.

c) _____ поток _____ обычно подключен к экрану компьютера.

d) Форматированный вывод выполняется функцией _____.

e) Строка управления форматом может содержать _____, _____, _____, _____ и _____.

f) Спецификаторы преобразования _____ и _____ используются для вывода десятичного целого со знаком.

g) Спецификаторы преобразования _____, _____ и _____ используются для вывода целых без знака в восьмеричном, десятичном и шестнадцатеричном форматах соответственно.

h) Модификаторы _____ и _____ помещаются перед спецификаторами преобразования целых для вывода целых значений типа **short** или **long**.

i) Спецификатор преобразования _____ используется для вывода значений с плавающей точкой в экспоненциальной нотации.

j) Модификатор _____ помещается перед любым спецификатором преобразования значений с плавающей точкой для вывода значений типа **long double**.

k) Спецификаторы преобразования **e**, **E** и **f** выводят значение с точностью, равной _____ цифрам справа от десятичной точки, если точность представления не указана.

l) Спецификаторы преобразования _____ и _____ используются для вывода соответственно строк и символов.

m) Все строки заканчиваются _____ символом.

n) Ширина поля и точность представления в спецификации преобразования функции **printf** могут задаваться целочисленным выражением при подстановке _____ в качестве значения ширины поля или значения точности и при включении этого целочисленного выражения в список аргументов функции.

o) Флаг _____ выравнивает выводимое значение по левому краю поля.

p) Флаг _____ выводит значение вместе со знаком «плюс» или знаком «минус».

q) Ввод с контролем формата выполняется функцией _____.

r) Для чтения заданных символов из потока и сохранения их в массиве символов используется _____.

s) Спецификатор преобразования _____ может использоваться для ввода восьмеричных, десятичных и шестнадцатеричных целых с необязательным знаком.

t) Спецификатор преобразования _____ может использоваться для ввода значений типа **double**.

u) _____ используется для чтения данных их входного потока и отбрасывания их без присваивания переменной.

v) Чтобы показать, что из входного потока должно быть введено определенное число символов или цифр, в спецификации преобразования **scanf** может использоваться _____.

9.2. Найдите ошибку в следующих операторах и объясните, как ее можно исправить.

a) Следующий оператор должен напечатать символ **'c'**

```
printf("%s\n", 'c');
```

b) Следующий оператор должен напечатать **9.375%**.

```
printf("%.3f%", 9.375);
```

c) Следующий оператор должен напечатать первый символ строки **«Monday»**

```
printf("%c\n", "Monday");
```

d) `printf(""A string in quotes"");`

e) `printf(%d%d, 12, 20);`

f) `printf("%c", "x");`

g) `printf("%s\n", 'Richard');`

9.3. Напишите операторы для выполнения следующих действий:

a) Напечатать **1234** с выравниванием по правому краю поля шириной **10** позиций.

b) Напечатать **123.456789** в экспоненциальной нотации со знаком (+ или -) и с точностью **3** разряда.

c) Ввести в переменную **number** значение типа **double**.

d) Напечатать **100** в восьмеричном формате с префиксом **0**.

e) Ввести строку в массив символов **string**.

f) Ввести символы в массив **n** до первого встретившегося символа, не относящегося к цифрам.

g) Используйте целые переменные **x** и **y** для задания ширины поля и точности представления при выводе значения **87.4573** типа **double**.

h) Введите значение в виде **3.5%**. Сохраните число в переменной **percent** типа **float**, отбросив при вводе символ % из входного потока. Не используйте символ подавления присваивания.

i) Напечатайте **3. 333333** как значение типа **long double** со знаком (+ или -) в поле шириной **20** символов и с точностью **3**.

Ответы на упражнения для самоконтроля

9.1. a) потоков. b) Стандартный, ввода. c) Стандартный, вывода. d) **printf**. e) спецификаторы преобразования, флаги, ширину полей, точность представления, литеральные символы. f) **d**, **i**. g) **o**, **u**, **x** (или **X**). h) **h**, **l**. i) **e** (или **E**). j) **L**. k) 6. l) **s**, c. m) нулевым (\0). n) звез-

дочки (*). o) – (минус). p) + (плюс). q) **scanf**. r) набор сканирования. s) **i**. t) **le**, **lE**, **lf**, **lg** или **lG**. u) Символ подавления присваивания (*). v) ширина поля.

9.2. a) Ошибка: спецификатор преобразования **s** предполагает передачу аргумента типа указатель на **char**.

Исправление: для вывода символа **c** используйте спецификатор преобразования **%c** или замените **'c' на «c»**.

b) Ошибка: попытка вывести литеральный символ % не используя спецификатор преобразования **%%**.

Исправление: используйте **%%** для вывода литерала %.

c) Ошибка: спецификатор преобразования **c** предполагает передачу аргумента типа **char**.

Исправление: чтобы вывести первый символ «Monday», используйте спецификацию преобразования **%1s**.

d) Ошибка: попытка вывести литеральный символ «не используя esc-последовательности \».

Исправление: замените обе внутренние кавычки на \».

e) Ошибка: строка управления форматом не заключена в двойные кавычки.

Исправление: заключите **%d%d** в двойные кавычки.

f) Ошибка: символ **x** заключен в двойные кавычки.

Исправление: символьные константы, выводящиеся со спецификацией преобразования **%c**, должны заключаться в одинарные кавычки.

g) Ошибка: выводимая строка заключена в одинарные кавычки.

Исправление: для вывода строки используйте двойные кавычки вместо одинарных.

9.3. a) `printf("%10d\n", 1234);`

b) `printf("%+.3e\n", 123.456789);`

c) `scanf("%lf, &number);`

d) `printf("%#o\n", 100);`

e) `scanf("%s", string);`

f) `scanf("%[^0123456789]", n);`

g) `printf ("%*.*f\n, x, y, 87.4573);`

h) `scanf("%f%%", &percent);`

i) `printf("%+20.3Lf\n", 3.333333);`

Упражнения

9.4. Выполните следующие задания, написав оператор **printf** или **scanf**:

a) Напечатайте целое значение без знака **40000**, с выравниванием по левому краю поля шириной **15** позиций и содержащее **8** цифр.

b) Введите шестнадцатеричное значение в переменную **hex**.

c) Напечатайте значение **200** со знаком и без знака.

d) Напечатайте значение **100** в шестнадцатеричном формате с префиксом **0x**.

e) Введите символы в массив **s** пока во входном потоке не встретится буква **p**.

f) Напечатайте значение **1.234** в поле из **9** цифр с заполняющими нулями.

g) Введите время в формате hh:mm:ss, запоминая численные значения в целых переменных **hour**, **minute** и **second** и отбрасывая двоеточия (:) из входного потока. Используйте символ подавления присваивания.

h) Введите строку **«characters»** из стандартного входного устройства. Сохраните строку в массиве символов **s**. Отбросьте кавычки из входного потока.

i) Введите время в формате **hh:mm:ss**, запоминая численные значения в целых переменных **hour**, **minute** и **second** и отбрасывая двоеточия (:) из входного потока. Не используйте символ подавления присваивания.

9.5. Покажите, что печатают следующие операторы. Если в операторе имеется ошибка, укажите, почему.

a) `printf("%-10d\n", 10000);`

b) `printf("%c\n", "This is a string");`

c) `printf("%*.*lf\n", 8, 3, 1024.987654);`

d) `printf("%#o\n%#X\n%#e\n", 17, 17, 1008.83689);`

e) `printf("% ld\n%+ld\n", 1000000, 1000000);`

f) `printf("%10.2E\n", 444.93738);`

g) `printf("%10.2g\n", 444.93738);`

h) `printf("%d\n", 10.987);`

9.6. Найдите ошибки в следующих фрагментах программ. Объясните, как их можно исправить.

a) `printf("%s\n", 'Happy Birthday');`

b) `printf("%c\n", 'Hello');`

c) `printf("%c\n", "This is a string");`

d) Следующий оператор должен напечатать **«Bon Voyage»**

```
printf(""%s"", "Bon Voyage");
```

e)
```
char day[] = "Sunday";
printf("%s\n", day[3]);
```

f) `printf('Enter your name: ');`

g) `printf(%f, 123.456);`

h) Следующий оператор должен напечатать символы **'O'** и **'K'**.

```
printf("%s%s\n", 'O', 'K');
```

i)
```
char s[10];
scanf("%c", s[7]);
```

9.7. Напишите программу, которая заполняет массив **number** из 10-ти элементов случайными целыми числами в диапазоне от 1 до 1000. Программа должна вывести каждое значение и текущее общее количество выведенных символов. Для этого используйте спецификацию преобразования **%n**, чтобы определить число выводимых символов для каждого текущего случайного значения. Каждый раз при печати текущего случайного значения печатайте общее число символов для всех выведенных значений, включая текущее. Вывод должен иметь следующий формат:

```
Value      Total characters
342        3
1000       7
963        10
6          11
и т.д.
```

9.8. Напишите программу для проверки различия между спецификаторами преобразования **%d** и **%i** при использовании их в операторе **scanf**. Для ввода и вывода значений используйте операторы

```
scanf("%i%d", &x, &y);
printf("%d %d\n", x, y);
```

Проверьте программу со следующим набором входных данных:

```
  10     10
 -10    -10
 010    010
0x10   0x10
```

9.9. Напишите программу, которая выводит значение указателя, используя все спецификаторы преобразования целых, а также спецификатор **%p**. Какой из них дает странные результаты? Какой из них вызывает ошибку? В каком формате выводит значение адреса на вашей системе спецификатор преобразования **%p**?

9.10. Напишите программу для проверки результатов вывода целого значения **12345** и значения с плавающей точкой **1.2345** в полях разной ширины. Что происходит, когда значения выводятся в поле, содержащее меньше цифр, чем выводимые значения?

9.11. Напишите программу, которая печатает значение **100.453627**, округленное до ближайшего целого, до десятых, сотых, тысячных и до десятитысячных.

9.12. Напишите программу, которая вводит строку с клавиатуры и определяет ее длину. Напечатайте строку в поле с шириной, равной удвоенной длине строки.

9.13. Напишите программу, которая преобразует целые значения температуры в шкале Фаренгейта в диапазоне от 0 до 212 градусов в значения температуры с плавающей точкой в шкале Цельсия с точностью 3 цифры. Для вычислений используйте формулу

```
celcius = 5.0 / 9.0 * (fahrenheit - 32);
```

Результат должен быть напечатан в два столбца шириной 10 символов с выравниванием по правому краю. Перед значением температуры в шкале Цельсия должен выводиться знак как для отрицательных температур, так и для положительных.

9.14. Напишите программу для проверки всех esc-последовательностей, приведенных на рис. 9.16. Для тех из них, которые перемещают курсор, напечатайте любой символ как до, так и после вывода esc-последовательности, чтобы было видно, куда переместился курсор.

9.15. Напишите программу, которая выясняет, можно ли вывести в операторе **printf** символ ? как литеральный символ, являющийся частью строки управления форматом, или нужно использовать esc-последовательность \?.

9.16. Напишите программу, которая вводит значение **437**, используя все доступные спецификаторы преобразования целых функции **scanf**. Напечатайте каждое введенное значение, используя все доступные спецификаторы преобразования целых.

9.17. Напишите программу, которая использует спецификаторы преобразования **e**, **f** и **g** для ввода значения **1.2345**. Напечатайте все введенные значения, чтобы удостовериться, что любой из этих спецификаторов преобразования может быть использован для ввода того же самого значения.

9.18. В некоторых языках программирования при вводе строк необходимо использовать одинарные *или* двойные кавычки. Напишите программу, которая читает три строки: **suzy**, **«suzy»** и **'suzy'**. Игнорируются ли те и другие кавычки при вводе в языке C или они читаются как часть строки?

9.19. Напишите программу, которая выясняет, можно ли со спецификатором преобразования **%c** в строке управления форматом оператора **printf** вывести символ ? как символьную константу — **'?'**, или нужно использовать символьную константу esc-последовательности **'\?'**.

9.20. Напишите программу, которая использует спецификатор преобразования **g** для вывода значения **9876.12345**. Напечатайте значение с точностью, изменяющейся от **1** до **9**.

глава

10

Структуры, объединения, операции с битами и перечисления



Цели

- Научиться создавать и использовать структуры, объединения и перечисления.
- Изучить передачу структур в функции по значению и по ссылке.
- Научиться работе с данными с помощью поразрядных операций.
- Научиться создавать битовые поля для компактного хранения данных.

Содержание

10.1. Введение
10.2. Описания структур
10.3. Инициализация структур
10.4. Доступ к элементам структур
10.5. Использование структур с функциями
10.6. Typedef
10.7. Пример: моделирование высокоэффективной тасовки и раздачи карт
10.8. Объединения
10.9. Поразрядные операции
10.10. Битовые поля
10.11. Перечислимые константы

Резюме • Распространенные ошибки программирования • Хороший стиль программирования • Советы по переносимости программ • Советы по повышению эффективности • Общие методические замечания • Упражнения для самоконтроля • Ответы на упражнения для самоконтроля • Упражнения

10.1. Введение

Структуры — это наборы (иногда их называют *агрегатами*) логически связанных переменных, объединенных под одним именем. В отличие от массивов, которые могут содержать элементы только одного типа, структуры могут состоять из переменных различных типов данных. Структуры часто используются, чтобы определить записи, которые должны сохраняться в файлах (см. главу 11, «Работа с файлами»). Указатели и структуры могут служить базой для создания более сложных структур данных, таких как связанные списки, очереди, стеки и деревья (см. главу 12, «Структуры данных»).

10.2. Описания структур

Структуры — это *производные типы данных*, они создаются из объектов других типов. Рассмотрим следующее описание структуры:

```
struct card {
   char *face;
   char *suit;
};
```

Ключевое слово **struct** определяет структуру. Идентификатор **card** является *именем-этикеткой* структуры. Имя-этикетка именует структуру, и используется совместно с ключевым словом **struct** для объявления переменных *типа структуры*. В данном примере тип структуры — **struct card**. Переменные, объявленные внутри скобок структуры, являются *элементами* структуры. Элементы одной структуры должны иметь уникальные имена, но две различных структуры могут содержать одинаковые имена, и это не вызовет никаких конфликтов (скоро мы увидим почему). Каждое определение структуры должно заканчиваться точкой с запятой.

Распространенная ошибка программирования 10.1

Забывают о точке с запятой, завершающей определение структуры.

Определение **struct card** содержит два элемента типа **char * — face** и **suit**. Элементы структуры могут быть переменными основных типов данных (то есть **int**, **float** и т.д.), или агрегатами, такими, как массивы или же другие структуры. Как мы видели в главе 6, все элементы массива должны быть одного типа. Элементы структуры, однако, могут относиться к различным типам данных. Например, **struct employee** может содержать элементы типа символьной строки для имени и фамилии, элемент типа **int** для возраста работника, элемент типа **char**, содержащий **'M'** или **'F'** для обозначения пола, элемент типа **float** для хранения данных о размерах почасовой оплаты работника и так далее. Элемент структуры не может быть структурой того же типа, в которой он содержится. Например, нельзя в определении **struct card** объявить переменную типа **struct card**. Однако ее можно объявить как указатель на тип структуры, в которую он входит. Структура, содержащая элемент — указатель на структуру того же типа, называется *структурой, ссылающейся на себя*. Ссылающиеся на себя структуры будут использованы в главе 12 для построения различных видов связанных структур данных.

Данное выше определение структуры не резервирует место под элементы структуры в памяти компьютера. Оно просто создает новый тип данных, который можно использовать для объявления переменных. Переменные структуры объявляются так же, как и переменные других типов. Объявление

```
struct card a, deck[52], *cPtr;
```

объявляет **a** — переменную типа **struct card**, **deck** — массив из 52 элементов типа **struct card** и **cPtr** — указатель на **struct card**. Можно объявить переменные данной структуры и по-другому, разместив разделенный запятыми список переменных между закрывающей скобкой определения структуры и точкой с запятой, завершающей ее определение. Например, указанное выше объявление, объединенное с определением структуры, будет выглядеть следующим образом:

```
struct card {
   char *face;
   char *suit;
} a, deck[52], *cPtr;
```

Имя-этикетка не является для структуры обязательным. Если определение структуры не содержит имя-этикетку, переменные для этой структуры могут быть объявлены только в определении структуры, но не отдельным объявлением.

Хороший стиль программирования 10.1

Давать каждой создаваемой структуре имя-этикетку. Имя-этикетка структуры позволит объявлять далее в программе новые переменные, принадлежащие к типу этой структуры.

Хороший стиль программирования 10.2

Осмысленный выбор имени-этикетки структуры позволит сделать программу самодокументированной.

Следующие операции над структурами являются единственно допустимыми: присваивание переменных-структур переменным того же типа, взятие адреса (&) структуры, обращение к элементам структуры (см. раздел 10.4), и применение операции **sizeof** для определения ее размера.

Распространенная ошибка программирования 10.2

Присваивают структуру одного типа структуре другого типа.

Нельзя сравнивать структуры, поскольку элементы структуры не обязательно хранятся в последовательных байтах памяти. Иногда в структурах бывают «дыры» из-за того, что компьютеры могут хранить специфические типы данных только в определенных позициях памяти, таких как граница полуслова, слова или двойного слова. Слово является стандартной единицей памяти, используемой для хранения данных в компьютере — обычно два или четыре байта. Рассмотрим следующее определение структуры, в которой объявляются переменные **sample1** и **sample2** типа **struct example**:

```
struct example {
   char c;
   int i;
} sample1, sample2;
```

Компьютер с размером слова в два байта может потребовать, чтобы каждый из элементов **struct example** был выровнен по границе слова, то есть по началу слова (выравнивание зависит от типа машины). На рис. 10.1 показан пример выравнивания памяти для переменной типа **struct example**, которой был присвоен символ '**a**' и целое **97** (показано двоичное представление этих величин). Если сохраняемые элементы выравниваются по границе слова, то при размещении в памяти переменных типа **struct example** образуется «дырка» размером в один байт (байт **1** на рисунке). И значение, находящееся в этой «дырке», не определено. И если значения элементов **sample1** и **sample2** на самом деле равны, сравнение структур совсем не обязательно покажет, что они

равны, так как в неопределенном первом байте совсем не обязательно будут содержаться одинаковые значения.

Распространенная ошибка программирования 10.3

Сравнение структур является синтаксической ошибкой, потому что на различных системах существуют отличающиеся друг от друга правила выравнивания.

Совет по переносимости программ 10.1

На разных машинах элементы данных отличаются размером, а кроме того, от типа машины зависит организация выравнивания памяти, поэтому и расположение структуры в памяти также зависит от типа машины.

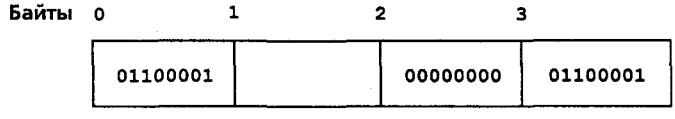


Рис. 10.1. Пример выравнивания памяти для переменной типа **struct example**, иллюстрирующий появление области памяти с неопределенным значением

10.3. Инициализация структур

Структуры можно инициализировать, как и массивы, используя список инициализации. Чтобы инициализировать структуру, после имени переменной в объявлении структуры ставится знак равенства, за которым следует помещенный в фигурные скобки, разделенный запятыми список инициализаторов. Например, объявление

```
struct card a = {"Three", "Hearts"};
```

создает переменную **a** типа **struct card** (структура определена выше) и присваивает элементу **face** значение **"Three"**, а элементу **suit** значение **"Hearts"**. Если инициализаторов в списке меньше, чем элементов в структуре, остальным элементам автоматически присваивается значение **0** (или **NULL**, если элемент — указатель). Переменным-структурам, объявленным вне определения любой функции (т.е. глобально) присваиваются **0** или **NULL**, если они явно не инициализированы во внешнем объявлении. Структуры можно инициализировать и с помощью оператора присваивания. При этом можно либо присвоить переменной-структуре переменную того же типа, либо присвоить значения отдельным элементам структуры.

10.4. Доступ к элементам структур

Для обращения к элементам структур используются две операции: *операция элемента структуры* (.), также называемая *операцией-точкой*, и *операция указателя структуры* (->), также называемая *операцией-стрелкой*. Операция элемента структуры обращается к элементу через имя переменной структуры. Например, для того чтобы напечатать элемент **suit** структуры **a** из предыдущего объявления, можно написать оператор

```
printf("%s", a.suit);
```

Операция указателя структуры, состоящая из знака минус (-) и знака больше (>) без пробела между ними, обращается к элементу через указатель структуры. Предположим, что переменная **aPtr** была объявлена как указатель на **struct card** и ей был присвоен адрес структуры **a**. Чтобы напечатать элемент **suit** структуры **a** при помощи указателя **aPtr**, напишите оператор

```
printf("%s", aPtr->suit);
```

Выражение **aPtr->suit** эквивалентно **(*aPtr).suit**, которое, обращаясь по адресу, содержащемуся в указателе, находит структуру **a** и обращается к элементу **suit**, используя операцию элемента структуры. Скобки в данном случае необходимы по той причине, что операция элемента структуры (.) имеет более высокий приоритет, чем операция косвенной адресации (*). Операции указателя структуры и элемента структуры, наряду с круглыми скобками и квадратными скобками ([]), использовавшимися для индексов массивов, являются операциями наивысшего приоритета и ассоциируются слева направо.

Хороший стиль программирования 10.3

Рекомендуется избегать использования одинаковых имен для элементов структур различного типа. Это не запрещено, но может привести к недоразумениям.

Хороший стиль программирования 10.4

Не рекомендуется вставлять пробелы слева и справа от знаков операций . и —>. Это поможет подчеркнуть, что выражение, в которых содержатся данные операции, по существу *являются именем одной переменой*.

Распространенная ошибка программирования 10.4

Вставляют пробел между составными частями — и > операции указателя структуры (или вставляют пробелы между составными частями других вводимых при помощи нескольких символов операций, за исключением **?:**).

Распространенная ошибка программирования 10.5

Пытаются сослаться на элемент структуры, используя только имя элемента.

Распространенная ошибка программирования 10.6

Пропускают скобки, когда ссылаются на элемент структуры, используя указатель и операцию элемента структуры (иными словами, запись ***aPtr.suit** является синтаксической ошибкой).

Программа на рис. 10.2 демонстрирует использование операций элемента структуры и указателя структуры. С помощью операции элемента структуры элементам **a** присваиваются значения **"Ace"** и **"Spades"** соответственно. Указателю **aPtr** присваивается адрес структуры **a**. Оператор **printf** печатает элементы структуры **a**, *используя операцию элемента структуры* с именем переменной **a**, операцию указателя структуры с указателем **aPtr**, и операцию элемента структуры с указателем **aPtr**, к которому применена операция косвенной адресации.

```
/* Применение операций элемента структуры
   и указателя структуры */
#include <stdio.h>

struct card {
   char *face;
   char *suit;
};

main()
{
   struct card a;
   struct card *aPtr;

   a.face ="Ace";
   a.suit ="Spades";
   aPtr =&a;
   printf("%s%s%s\n%s%s%s\n%s%s%s\n",
          a.face, " of ", a.suit,
          aPtr->face, " of ", aPtr->suit,
          (*aPtr).face, " of ", (*aPtr).suit);
   return 0;
}
```

```
Ace of Spades
Ace of Spades
Ace of Spades
```

Рис. 10.2. Использование операций элемента структуры и указателя структуры

10.5. Использование структур с функциями

Структуры могут передаваться функциям посредством передачи отдельных элементов структуры, передачи всей структуры или передачи указателя на структуру. Когда структуры или отдельные их элементы передаются функции, они передаются вызовом по значению. Поэтому вызванная функция не может изменять элементы структуры в вызывающей.

Чтобы осуществить вызов структуры по ссылке, необходимо передать адрес структуры. Массивы структур, как и все прочие массивы, вызываются по ссылке автоматически.

В главе 6 было отмечено, что, используя структуру, можно осуществить вызов массива по значению. Для вызова массива по значению создается структура, содержащая массив как элемент. Вызывая структуру по значению, мы и массив вызываем по значению.

Распространенная ошибка программирования 10.7

Полагая, что структуры, как и массивы, вызываются по ссылке автоматически, пытаются изменить значение структуры вызывающей функции в вызываемой.

Совет по повышению эффективности 10.1

Вызов структуры по ссылке более эффективен, чем вызов по значению (при котором происходит копирование всей структуры целиком).

10.6. Typedef

Ключевое слово **typedef** предоставляет программисту механизм для создания синонимов (или псевдонимов) для ранее определенных типов данных. Часто используют **typedef** для того, чтобы дать укороченное имя структурному типу. Например, оператор

```
typedef struct card Card;
```

определяет новый тип с именем **Card**, как синоним типа **struct card**. Пишущие на С часто используют **typedef**, чтобы определить тип структуры, при этом отпадает необходимость в имени-этикетке. Например, следующее определение

```
typedef struct {
   char *face;
   char *suit;
} Card;
```

создает тип **Card** без использования отдельного оператора **typedef**.

Хороший стиль программирования 10.5

Начинать имена, определяемые **typedef**, с прописной буквы, чтобы подчеркнуть тот факт, что эти имена являются синонимами имен других типов.

Теперь **Card** можно использовать для объявления переменных типа **struct card**. Объявление

```
Card deck [52];
```

описывает массив, состоящий из 52 структур **Card** (т.е. переменных типа **struct card**). Создание нового имени с помощью **typedef** не создает нового типа; **typedef** просто определяет новое имя для уже существующего типа, которое может использоваться как псевдоним последнего. Осмысленно выбранный псевдоним позволяет сделать программу самодокументированной. Например, прочитав предыдущее объявление, сразу можно догадаться, что «колода (deck) — это массив из 52 карт (cards)».

Часто **typedef** используется для присвоения псевдонимов основным типам данных. Например, программа, которой требуются четырехбайтовое целое, может использовать тип **int** в одной системе и тип **long** в другой. В программах, предназначенных для работы на разных машинах, часто дают четырехбайтовым целым специальный псевдоним, скажем, **Integer**. После этого достаточно изменить псевдоним **Integer** в одном месте программы, чтобы программа заработала на обеих системах.

Совет по переносимости программ 10.2

Использование **typedef** помогает облегчить адаптацию программы к различным аппаратно-программным платформам.

10.7. Пример: моделирование высокоэффективной тасовки и раздачи карт

Программа на рис. 10.3 основана на методе моделирования тасовки и раздачи карт, обсуждавшемся в главе 7. Программа представляет колоду карт как массив структур. Она использует высокоэффективные алгоритмы тасования и раздачи. Результат выполнения программы показан на рис. 10.4.

```
/* Программа тасовки и сдачи, использующая структуры */
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

struct card {
   char *face;
   char *suit;
};

typedef struct card Card;

void fillDeck(Card *, char *[], char *[]);
void shuffle(Card *);
void deal(Card *);

main()
{
   Card deck[52];
   char *face[] = {"Ace", "Deuce", "Three", "Four", "Five",
                   "Six", "Seven", "Eight", "Nine", "Ten",
                   "Jack", "Queen", "King"};
   char *suit[] = {"Hearts", "Diamonds", "Clubs", "Spades"};

   srand(time(NULL));

   fillDeck(deck, face, suit);
   shuffle(deck);
   deal(deck);
   return 0;
}

void fillDeck(Card *wDeck, char *wFace[], char *wSuit[])
{
   int i;

   for (i = 0; i < 52; i++) {
      wDeck[i].face = wFace[i % 13];
      wDeck[i].suit = wSuit[i / 13];
   }
}

void shuffle(Card *wDeck)
{
   int i,j;
   Card temp;
```

Рис. 10.3. Высокоэффективное моделирование тасовки и раздачи карт (часть 1 из 2)

```
   for (i = 0; i < 52; i++) {
      j = rand()0 % 52;
      temp = wDeck[i];
      wDeck[i] = wDeck[j];
      wDeck[j] = temp;
   }
}

void deal(Card *wDeck)
{
   int i;
   for (i = 0; i < 52; I++)
      printf("%5s of %-8s%c", wDeck[i].face, wDeck[i].suit,
             (i + 1) % 2 ? '\t' : '\n');
}
```

Рис. 10.3. Высокоэффективное моделирование тасовки и раздачи карт (часть 2 из 2)

```
Eight of Diamonds          Ace of Hearts
Eight of Clubs            Five of Spades
Seven of Hearts          Deuce of Diamonds
  Ace of Clubs             Ten of Diamonds
Deuce of Spades            Six of Diamonds
Seven of Spades          Deuce of Clubs
 Jack of Clubs             Ten of Spades
 King of Hearts           Jack of Diamonds
Three of Hearts          Three of Diamonds
Three of Clubs            Nine of Clubs
  Ten of Hearts          Deuce of Hearts
  Ten of Clubs           Seven of Diamonds
  Six of Clubs           Queen of Spades
  Six of Hearts          Three of Spades
 Nine of Diamonds            Ace of Diamonds
 Jack of Spades            Five of Clubs
 King of Diamonds          Seven of Clubs
 Nine of Spades             Four of Hearts
  Six of Spades           Eight of Spades
Queen of Diamonds           Five of Diamonds
  Ace of Spades             Nine of Hearts
 King of Clubs              Five of Hearts
 King of Spades             Four of Diamonds
Queen of Hearts          Eight of Hearts
 Four of Spades             Jack of Hearts
 Four of Clubs           Queen of Clubs
```

Рис. 10.4. Результат тасовки и раздачи карт

Программа работает следующим образом. Функция **fillDeck** инициализирует массив **Card** в порядке от туза до короля для каждой масти. Массив **Card** передается функции **shuffle**, в которой реализован эффективный алгоритм тасования. Функция получает в качестве аргумента массив из 52 структур **Card**. Она обращается к каждой из 52 карт (индексы массива от 0 до 51), используя структуру **for**. Для каждой карты случайным образом выбирается число между 0 и 51. Следующим шагом текущая структура **Card** и случайно выбранная структура меняются в массиве местами. Полностью 52 замены выполняются

за один проход всего массива, и массив структур **Card** перетасован! Этот алгоритм свободен от порока, связанного с бесконечной отсрочкой, возникающей в алгоритме тасования из главы 7. Так как структуры **Card** уже переставлены местами, алгоритм раздачи, реализованный в функции **deal**, требует всего лишь одного прохода по массиву, чтобы раздать тасованные карты.

Распространенная ошибка программирования 10.8

Забывают индекс массива при ссылке на отдельные структуры в массиве структур.

10.8. Объединения

Объединение — производный тип данных, подобный структуре, элементы которого разделяют одну и ту же область памяти. На различных этапах выполнения программы одни переменные могут оказаться невостребованными, в то время как другие, наоборот, используются только в этой части программы, поэтому объединения экономят пространство вместо того, чтобы впустую тратить память на не использующиеся в данный момент переменные. Элементы объединения могут принадлежать к любому типу. Число байтов, используемое для хранения объединения, должно быть, по крайней мере, достаточным для хранения наибольшего из элементов. В большинстве случаев объединения содержат два или более типа данных. Ссылаться в данный момент времени можно только на один элемент и, соответственно, только один тип данных. Задача программиста — обеспечить, чтобы на данные, хранящиеся в объединении, ссылались как на данные соответствующего типа.

Распространенная ошибка программирования 10.9

Обращаются к данным, хранящимся в объединении, как к данным одного типа, в то время как хранящиеся в данный момент в объединении данные принадлежат другому типу, что приводит к появлению логической ошибки.

Совет по переносимости программ 10.3

Если сохранять данные в объединении, как принадлежащие одному типу, а обращаться к ним в дальнейшем как к данным другого типа, то результат будет зависеть от реализации системы.

Объединение объявляется с помощью ключевого слова **union**. Формат объявления тот же, что и в случае структуры. Объявление **union**

```
union number {
   int x;
   float y;
};
```

означает, что **number** является типом **union** с элементами **int x** и **float y**. Определение объединения обычно располагается в программе перед функцией **main**, поэтому определение может использоваться для объявления переменных во всех функциях программы.

Общее методическое замечание 10.1

Как и в случае объявления **struct**, объявление **union** просто создает новый тип. Размещение объявления **union** или **struct** вне функции не создает глобальной переменной.

Над объединениями можно выполнять следующие операции: присваивание объединения другому объединению того же типа, взятие адреса (&), доступ к элементам объединения с использованием операций элемента структуры и указателя структуры. Объединения не могут сравниваться по тем же самым причинам, что и структуры.

При объявлении объединение можно инициализировать только значениями того же самого типа, что и его первый элемент. Например, в представленном выше объединении объявление

```
union number value = {10};
```

является допустимой инициализацией переменной **value**, потому что объединение инициализировано как **int**, однако объявление следующего вида будет ошибочным:

```
union number value = {1.43};
```

Распространенная ошибка программирования 10.10

Сравнение объединений является синтаксической ошибкой, поскольку в разных системах существуют различные требования на выравнивание.

Распространенная ошибка программирования 10.11

Инициализация объединения в объявлении значением, имеющим тип, отличный от типа первого элемента объединения.

Совет по переносимости программ 10.4

Объем памяти, необходимый для хранения объединения, зависит от типа компьютера.

Совет по переносимости программ 10.5

Некоторые объединения невозможно просто перенести на другую компьютерную систему. Является ли объединение переносимым или нет, зависит часто от требований на выравнивание памяти для типов данных элементов объединения в данной системе.

Совет по повышению эффективности 10.2

Объединения позволяют сэкономить память.

Программа на рис. 10.5 объявляет переменную **value** типа **union number** и отображает значения, хранящиеся в объединении, как тип **int** и как **float**. Результат выполнения программы зависит от реализации. Вывод программы свидетельствует, что внутреннее представление значения **float** может значительно отличаться от представления **int**.

```
/* Пример объединения */
#include <stdio.h>

union number {
   int  x;
   float y;
};

main()
{
   union number value;

   value.x= 100;
   printf("%s\n%s\n%s%d\n%s%f\n\n",
          "Put a value in the integer member",
          "and printf both members.",
          "int:   ", value.x,
          "float: ", value.y);

   value.y= 100.0;
   printf("%s\n%s\n%s%d\n%s%f\n",
          "Put a value in the floating member",
          "and printf both members.",
          "int:   ", value.x,
          "float: ", value.y);
  return 0;
}
```

```
Put a value in the integer member
and print both members/
int:   100
float: 0.000000

Put a value in the floating member
and print both members.
int:   17096
float: 100.000000
```

Рис. 10.5. Вывод на печать значения элемента объединения в формате каждого из типов данных

10.9. Поразрядные операции

Все данные представляются внутри компьютера как последовательности битов. Каждый бит может принимать значения **0** или **1**. В большинстве систем последовательность из 8 бит образует байт — стандартную единицу памяти для переменной типа **char**, для хранения других типов данных используется большее число байтов. Поразрядные операции используются для операций над битами целочисленных операндов (**char**, **short**, **int** и **long**; как **signed**, так и **unsigned**). В поразрядных операциях обычно используются беззнаковые целые.

Совет по переносимости программ 10.6

Поразрядные операции с двоичными данными являются машинно-зависимыми.

Заметим, что в поразрядных операциях, обсуждаемых в этом разделе, используется двоичное представление целых операндов. Более подробное описание операций в двоичной системе счисления приведено в приложении Д, «Системы счисления». Кроме того, программы разделов 10.9 и 10.10 были протестированы на Apple Macintosh с использованием Think C и на PC-совместимой машине с использованием Borland C++. В обеих системах используются 16-битные (2-байтовые) целые. По причине того, что выполнение поразрядных операций зависит от типа машины, эти программы могут и не работать на вашей системе.

К поразрядным относятся следующие операции: *поразрядное И (&), поразрядное включающее ИЛИ (|), поразрядное исключающее ИЛИ (^), сдвиг влево (<<), сдвиг вправо (>>)* и *дополнение (~)*. В операциях поразрядного И, поразрядного включающего ИЛИ и поразрядного исключающего ИЛИ два операнда сравниваются побитно. Операция поразрядного И устанавливает каждый бит значения результата равным 1, если соответствующие биты каждого из операндов равны 1. Операция поразрядного включающего ИЛИ устанавливает каждый бит значения результата равным 1, если соответствующий бит в одном (или в обоих) операндах равен 1. Операция поразрядного исключающего ИЛИ устанавливает каждый бит значения результата равным 1, если соответствующий бит равен 1 только в одном из операндов. Операция сдвига влево сдвигает биты в левом операнде влево на число бит, заданное в правом операнде. Операция сдвига вправо сдвигает биты в левом операнде вправо на число бит, заданное в правом операнде. В результате выполнения операции поразрядного дополнения все биты, равные 0, устанавливаются равными 1, а все биты, равные 1, устанавливаются равными 0. Детальное обсуждение каждой из поразрядных операций дано в следующих ниже примерах. Сводная таблица поразрядных операций приведена на рис. 10.6.

Операции	Описание
& поразрядное И	Бит результата устанавливается равным **1**, если соответствующие биты каждого из операндов равны **1**.
| поразрядное включающее ИЛИ	Бит результата устанавливается равным **1**, если хотя бы один соответствующий бит двух операндов равен **1**.
^ поразрядное исключающее ИЛИ	Бит результата устанавливается равным **1**, если только один соответствующий бит двух операндов равен **1**.
<< сдвиг влево	Сдвигает биты первого операнда влево на число бит, задаваемое вторым операндом; справа заполняется нулевыми битами.
>> сдвиг вправо	Сдвигает биты первого операнда вправо на число битов, задаваемое вторым операндом; метод заполнения слева зависит от типа данных и конкретной системы.
~ дополнение	Все биты, равные **0**, устанавливает равными **1**, а все биты, равные **1**, устанавливает равными **0**.

Рис. 10.6. Поразрядные операции

При использовании поразрядных операций полезно выводить значения операндов в двоичном виде, чтобы получить наглядное представление о результатах их выполнения. Программа на рис. 10.7 выводит на печать целое без знака в двоичном представлении группами по восемь бит. Функция **displayBits** применяет операцию поразрядного И к переменным **value** и **displayMask**. Часто операция

поразрядного И используется с операндом, называемым *маской*. Маска — это целое значение, определенные биты которого установлены равными **1**. Маски используются для того, чтобы спрятать некоторые биты числа. В функции **displayBits** маске **displayMask** присваивается значение **1 << 15 (10000000 00000000)**. Операция сдвига влево сдвигает в **displayMask** величину **1** из бита младшего разряда (крайнего справа) в бит старшего разряда (крайний слева), и заполняет нулями биты справа. Оператор

```
putchar(value & displayMask ? '1' : '0');
```

определяет, что требуется напечатать для текущего крайнего слева бита переменой **value**, **1** или **0**. Предположим, переменная **value** равна **65000 (11111101 11101000)**. Когда **value** и **displayMask** комбинируются посредством &, все биты переменной **value**, за исключением старшего, становятся «маскированными» (скрытыми), потому что любой бит, подвергнутый операции И с **0**, дает **0**. Если крайний слева бит равен **1**, **value & displayMask** имеет значение **1** и на печать выводится **1**, в противном случае на печать выводится **0**. Переменная **value** затем сдвигается влево на один бит при помощи выражения **value <<= 1** (это эквивалентно выражению **value = value << 1**). И эти шаги повторяются для каждого бита в переменной **value** типа **unsigned**. На рис. 10.8 дана сводная таблица значений объединения двух битов посредством поразрядной операции И.

```
/* Побитная печать беззнакового целого */
#include <stdio.h>

main()
{
   unsigned x;
   void displayBits(unsigned);

   printf("Enter an unsigned integer: ");
   scanf("%u", &x);
   displayBits(x);
   return 0;
}

void displayBits(unsigned value)
{
   unsigned c, displayMask = 1 << 15;

   printf("%7u = ", value);
   for (c = 1; c <= 16; c++) {
      putchar(value & displayMask ? '1' : '0');
      value <<= 1;

      if (c % 8 == 0)
         putchar(' ');
   }
   putchar('\n');
}
```

```
Enter an unsigned integer: 65000
  65000 = 11111101 11101000
```

Рис. 10.7. Вывод на печать целого без знака в двоичном виде

Bit1	Bit2	Bit1 & Bit2
0	0	0
1	0	0
0	1	0
1	1	1

Рис. 10.8. Результаты объединения двух битов посредством поразрядной операции И (**&**)

Распространенная ошибка программирования 10.12

Использование логической операции И (**&&**) вместо поразрядной операции (**&**) и наоборот.

Программа на рис. 10.9 иллюстрирует выполнение поразрядных операций И, включающего ИЛИ, исключающего ИЛИ и операции поразрядного дополнения. Программа вызывает функцию **displayBits** для вывода на печать целых величин типа **unsigned**. Результаты программы показаны на рис. 10.10.

```
/* Применение операций поразрядного AND, включающего OR,
   исключающего OR и поразрядного дополнения */

#include <stdio.h>

void displayBits(unsigned);

main()
{
   unsigned number1, number2, mask, setBits;

   number1 = 65535;
   mask = 1;
   printf("The result of combining the following\n");
   displayBits(number1);
   displayBits(mask);
   printf("using the bitwise AND operator & is\n");
   displayBits(number1 & mask);

   number1 = 15;
   setBits = 241;
   printf("\nThe result of combining the following\n");
   displayBits(number1);
   displayBits(setBits);
   printf("using the bitwise inclusive OR operator | is\n");
   displayBits(number1 | setBits);

   number1 = 139;
   number2 = 199;
   printf("\nThe result  of combining the following\n");
   displayBits(number1);
```

Рис. 10.9. Использование операций поразрядных И, включающего ИЛИ, исключающего ИЛИ и дополнения (часть 1 из 2)

```
   displayBits(number2);
   printf("using the bitwise exclusive OR operator ^ is\n");
   displayBits(number1 ^ number2);

   number1 = 21845;
   printf("\nThe one's complement of\n");
   displayBits(number1);
   printf("is\n");
   displayBits(~number1);
   return 0;
}

void displayBits(unsigned value)
{
   unsigned c, displayMask = 1 << 15;

   printf("%7u = ", value);

   for (c = 1; c <= 16; c ++) {
      putchar(value & displayMask ? '1' : '0');
      value <<=1;

      if (c % 8 ==0)
         putchar(' ');
   }
   putchar('\n');
}
```

Рис. 10.9. Использование операций поразрядных И, включающего ИЛИ, исключающего ИЛИ и дополнения (часть 2 из 2)

```
The result of combining the following
  65535 = 11111111 11111111
      1 = 00000000 00000001
using the bitwise AND operator & is
      1 = 00000000 00000001

The result of combining the following
     15 = 00000000 00001111
    241 = 00000000 11110001
using the bitwise inclusive OR operator | is
    255 = 00000000 11111111

The result of combining the following
    139 = 00000000 10001011
    199 = 11000111
using the bitwise exclusive OR operator ^ is
     76 = 00000000 01001100

The one's complement of
  21845 = 01010101 01010101
is
  43690 = 10101010 10101010
```

Рис. 10.10. Результат выполнения программы, представленной на рис. 10.9

В программе на рис. 10.9 целой переменной **mask** присваивается значение **1 (00000000 00000001)**, а переменной **number1** значение **65535 (11111111 11111111)**. Переменные **mask** и **number1** объединяются в выражении **number1 & mask** посредством операции поразрядного И (&). В результате получаем **00000000 00000001**. Все биты, за исключением бита младшего разряда в переменной **number1** маскированы (скрыты) после применения операции И с переменной **mask**.

Поразрядная включающая операция ИЛИ используется для того, чтобы установить в операнде заданные биты равными **1**. В программе на рис. 10.9 переменной **number1** присваивается значение **15 (00000000 00001111)**, а переменной **setBits — 241 (00000000 11110001)**. Когда **number1** и **setBits** объединяются операцией поразрядного ИЛИ в выражении **number1 | setBits**, в результате получается **255 (00000000 11111111)**. На рис. 10.11 дана сводная таблица значений объединения двух битов при помощи поразрядной операции включающего ИЛИ.

Bit1	Bit2	Bit1 | Bit2
0	0	0
1	0	1
0	1	1
1	1	1

Рис. 10.11. Результаты объединения двух бит посредством операции поразрядного включающего ИЛИ |

Распространенная ошибка программирования 10.13

Использование логической операции ИЛИ (||) вместо поразрядной операции (|) и наоборот.

Поразрядная исключающая операция ИЛИ (^) устанавливает бит результата равным **1**, если только один из соответствующих битов в двух операндах равен 1. В программе на рис. 10.9 переменным **number1** и **number2** присваиваются соответственно значения **139 (00000000 10001011)** и **199 (00000000 11000111)**. Комбинируются эти переменные при помощи операции исключающего ИЛИ в выражении **number1 ^ number2**, и получается **00000000 01001100**. На рис. 10.12 дана сводная таблица результатов объединения двух битов при помощи поразрядной операции исключающего ИЛИ.

Bit1	Bit2	Bit1 ^ Bit2
0	0	0
1	0	1
0	1	1
1	1	0

Рис. 10.12. Результаты объединения двух бит посредством операции поразрядного исключающего ИЛИ (^)

В результате выполнения операции поразрядного дополнения (~) всем битам операнда, имевшим значение **1**, присваивается значение **0**, а всем битам, имевшим значение **0**, соответственно **1** — по-другому эту операцию называют *дополнением до единицы*. На рис. 10.9 переменной **number1** присваивается значение **21845 (01010101 01010101)**. В результате вычисления выражения **~number1** получаем **(10101010 10101010)**.

Программа на рис. 10.13 представляет собой пример использования операций сдвига влево (<<) и сдвига вправо (>>). Функция **displayBits** используется для вывода на печать целых величин типа **unsigned**.

Операция сдвига влево (<<) смещает биты левого операнда влево на число бит, определенное в правом операнде. Биты, освобождаемые справа, заменяются нулями; «сбрасываемые» влево биты теряются. В программе на рис. 10.13 переменной **number1** присваивается значение **960 (00000011 11000000)**. В результате выполнения задаваемой выражением **number1 << 8** операции сдвига влево на 8 бит переменная **number1** получает значение **49152 (11000000 00000000)**.

```
/* Применение операций сдвига */
#include <stdio.h>

void displayBits(unsigned);

main()
{
   unsigned number1 = 960;

   printf("\The result of left shifting\n");
   displayBits(number1);
   printf("8 bit positions using the ");
   printf("left shift opertor << is\n");
   displayBits(number1 << 8);

   printf("\nThe result of right shifting\n");
   displayBits(number1);
   printf("8 bit positions using the ");
   printf("right shift operator >> is\n");
   displayBits(number1 >> 8);
   return 0;
}

void displayBits(unsigned value)
{
   unsigned c, displayMask = 1 << 15;

   printf("%7u = ", value);

   for (c = 1; c <= 16; c++) {
      putchar(value & displayMask ? '1' : '0');
      value <<=1;

      if (c % 8 == 0)
         putchar(' ');
   }

  putchar('\n');
}
```

Рис. 10.13. Использование операций поразрядного сдвига (часть 1 из 2)

```
The result of left shifting
    960 = 00000011 11000000
8 bit positions using the left shift operator << is
  49152 = 11000000 00000000

The result of right shifting
    960 = 00000011 11000000
8 bit positions using the right shift operator >> is
      3 = 00000000 00000011
```

Рис. 10.13. *Использование операций поразрядного сдвига (часть 2 из 2)*

Операция сдвига вправо (>>) смещает биты левого операнда вправо на число бит, определенное в правом операнде. Выполнение сдвига вправо для целого типа **unsigned** приводит к тому, что биты, освобождаемые слева, заменяются нулями; «сбрасываемые» вправо биты теряются. В программе на рис. 10.13 в результате выполнения задаваемой выражением **number1 >> 8** операции сдвига вправо на 8 бит переменная **number1** получает значение **3 (00000000 00000011)**.

Распространенная ошибка программирования 10.14

Значение, получаемое в результате сдвига, становится неопределенным, если правый операнд имеет отрицательное значение или его значение больше, чем число бит, в которых хранится левый операнд.

Совет по переносимости программ 10.7

Реализация операции сдвига вправо зависит от типа машины. Сдвиг вправо для отрицательного целого со знаком заполняет освобождаемые биты **0**-ми на одних машинах и **1**-ми на других.

Каждая поразрядная операция (исключая поразрядную операцию дополнения) имеет соответствующие знаки операции присваивания. Эти *знаки операций поразрядного присваивания* показаны на рис. 10.14 и используются точно так же, как и введенные в главе 3 знаки арифметического присваивания.

Знаки операций поразрядного присваивания	
&=	Операция присваивания поразрядного И.
|=	Операция присваивания поразрядного включающего ИЛИ.
^=	Операция присваивания поразрядного исключающего ИЛИ.
<<=	Операция присваивания сдвига влево.
>>=	Операция присваивания сдвига вправо.

Рис. 10.14. Знаки операций поразрядного присваивания

На рис. 10.15 показаны приоритет и ассоциативность различных операций, введенных начиная с первой страницы книги и оканчивая этими строками. Они расположены сверху вниз в порядке убывания приоритета.

Операции	Ассоциативность	Тип
() [] . ->	слева направо	наивысший приоритет
+ - ++ -- ! (type) & * ~ sizeof	справа налево	одноместные
* / %	слева направо	мультипликативные
+ -	слева направо	аддитивные
<< >>	слева направо	сдвига
< <= > >=	слева направо	отношения
== !=	слева направо	равенства
&	слева направо	поразрядное И
^	слева направо	исключающее ИЛИ
|	слева направо	поразрядное ИЛИ
&&	слева направо	логическое И
||	слева направо	логическое ИЛИ
?:	справа налево	условная
= += -= *= /= %= &= |= ^= <<= >>=	справа налево	присваивания
,	слева направо	запятая

Рис. 10.15. Приоритет и ассоциативность операций

10.10. Битовые поля

В C существует возможность задать число битов для хранения элемента структуры или объединения типа **unsigned** или **int** посредством определения *битовых полей*. Битовые поля позволяют лучше использовать память, храня данные в минимально требуемом количестве бит. Элементы — битовые поля *должны* быть объявлены как **int** или **unsigned**.

Совет по повышению эффективности 10.3

Битовые поля экономят память.

Рассмотрим следующее определение структуры:

```
struct bitCard {
   unsigned face : 4;
   unsigned suit : 2;
   unsigned color : 1;
};
```

Оно содержит три битовых поля типа **unsigned** — **face**, **suit** и **color**, — используемых для представления колоды из 52 карт. Битовое поле объявляется с помощью двоеточия (**:**) и целой константы, задающей *ширину* поля, то есть число бит, выделяемых для хранения элемента, помещаемых после имен элементов типа **unsigned** или **int**. Константа, задающая ширину, должна быть це-

лым числом, значение которого можно выбрать от 0 до полного числа бит, используемого для хранения **int** в вашей системе. Наши примеры проверялись на компьютерах с 2-байтовыми (16-битными) целыми.

Данное выше определение структуры показывает, что элемент **face** хранится в 4-х битах, элемент **suit** в 2-х битах, а элемент **color** в 1-м бите. Число отводимых бит определяется диапазоном значений для каждого элемента структуры. В элементе **face** хранятся значения величин от **0** (туз) до **12** (король) — в четырех битах могут храниться значения от 0 до 15. В элементе **suit** хранятся значения величин от **0** до **3** (**0** = бубны, **1** = червы, **2** = трефы, **3** = пики) — в двух битах могут храниться значения от 0 до 3. И наконец, в элементе **color** хранятся значения цвета **0** (красная) или **1** (черная) — в одном бите могут храниться значения 0 или 1.

Программа на рис. 10.16 (результат ее выполнения показан на рис. 10.17) создает массив **deck**, содержащий 52 структуры **struct bitCard**. Функция **fillDeck** размещает 52 карты в массиве **deck**, а функция **deal** распечатывает карты. Заметим, что обращение к битовым полям происходит точно так же, как и к любым другим элементам структуры. Элемент **color** включен для того, чтобы иметь возможность отображать цвет карты в системах, которые позволяют выводить цвета.

Допускается определение *неименованного битового поля*; такое поле используется как *заполнитель* структуры. Например, определение структуры

```
struct example {
   unsigned a : 13;
   unsigned   : 3;
   unsigned b : 4;
};
```

использует неименованное поле шириной в 3 бита — в этих трех битах невозможно что-либо сохранить. Элемент **b** (в случае, если ваш компьютер имеет двухбайтовые слова) будет размещен в другой ячейке памяти.

Неименованное битовое поле нулевой ширины служит для выравнивания следующего битового поля по границе новой ячейки памяти. Например, определение структуры

```
struct example {
   unsigned a : 13;
   unsigned   : 0;
   unsigned b : 4;
};
```

объявляет битовое поле без имени длиной **0** бит для того, чтобы перескочить через оставшиеся биты (столько, сколько их будет) ячейки памяти, в которой хранится **a**, и выровнять **b** по границе следующей ячейки.

Совет по переносимости программ 10.8

Операции с битовыми полями зависят от типа машины. Некоторые компьютеры, например, позволяют битовым полям пересекать границы слов, в то время как другие нет.

Распространенная ошибка программирования 10.15

Попытка обратиться к отдельным битам битового поля так, как если бы они были элементами массива. Битовые поля — это вовсе не «массивы бит».

```
/* Пример применения битовых полей */
#include <stdio.h>

struct bitCard {
   unsigned face : 4;
   unsigned suit : 2;
   unsigned color : 1;
};

typedef struct bitCard Card;

void fillDeck(Card *);
void deal(Card *);

main()
{
   Card deck[52];

   fillDeck(deck);
   deal(deck);

   return 0;
}

void fillDeck(Card *wDeck)
{
   int i;

   for (i = 0; i <= 51; i++) {
      wDeck[i].face = i % 13;
      wDeck[i].suit = i / 13;
      wDeck[i].color = i / 26;
   }
}

/* Функция deal печатает колоду в две колонки */
/* Колонка 1 содержит карты 0-25, индекс k1 */
/* Колонка 2 содержит карты 26-51, индекс k2 */

void deal(Card *wDeck)
{
   int k1, k2;

   for (k1 = 0, k2 = k1 + 26; k1 <= 25; k1++, k2++) {
      printf("Card:%3d  Suit:%2d  Color:%2d   ",
             wDeck[k1].face, wDeck[k1].suit, wDeck[k1].color);
      printf("Card:%3d  Suit:%2d  Color:%2d\n",
             wDeck[k2].face, wDeck[k2].suit, wDeck[k2].color);
   }
}
```

Рис. 10.16. Использование битовых полей для хранения колоды карт

Распространенная ошибка программирования 10.16

Попытка взять адрес битового поля (операция **&** неприменима к битовым полям, так как они не имеют адреса).

```
Card:  0 Suit: 0 Color: 0   Card:  0 Suit: 2 Color: 1
Card:  1 Suit: 0 Color: 0   Card:  1 Suit: 2 Color: 1
Card:  2 Suit: 0 Color: 0   Card:  2 Suit: 2 Color: 1
Card:  3 Suit: 0 Color: 0   Card:  3 Suit: 2 Color: 1
Card:  4 Suit: 0 Color: 0   Card:  4 Suit: 2 Color: 1
Card:  5 Suit: 0 Color: 0   Card:  5 Suit: 2 Color: 1
Card:  6 Suit: 0 Color: 0   Card:  6 Suit: 2 Color: 1
Card:  7 Suit: 0 Color: 0   Card:  7 Suit: 2 Color: 1
Card:  8 Suit: 0 Color: 0   Card:  8 Suit: 2 Color: 1
Card:  9 Suit: 0 Color: 0   Card:  9 Suit: 2 Color: 1
Card: 10 Suit: 0 Color: 0   Card: 10 Suit: 2 Color: 1
Card: 11 Suit: 0 Color: 0   Card: 11 Suit: 2 Color: 1
Card: 12 Suit: 0 Color: 0   Card: 12 Suit: 2 Color: 1
Card:  0 Suit: 1 Color: 0   Card:  0 Suit: 3 Color: 1
Card:  1 Suit: 1 Color: 0   Card:  1 Suit: 3 Color: 1
Card:  2 Suit: 1 Color: 0   Card:  2 Suit: 3 Color: 1
Card:  3 Suit: 1 Color: 0   Card:  3 Suit: 3 Color: 1
Card:  4 Suit: 1 Color: 0   Card:  4 Suit: 3 Color: 1
Card:  5 Suit: 1 Color: 0   Card:  5 Suit: 3 Color: 1
Card:  6 Suit: 1 Color: 0   Card:  6 Suit: 3 Color: 1
Card:  7 Suit: 1 Color: 0   Card:  7 Suit: 3 Color: 1
Card:  8 Suit: 1 Color: 0   Card:  8 Suit: 3 Color: 1
Card:  9 Suit: 1 Color: 0   Card:  9 Suit: 3 Color: 1
Card: 10 Suit: 1 Color: 0   Card: 10 Suit: 3 Color: 1
Card: 11 Suit: 1 Color: 0   Card: 11 Suit: 3 Color: 1
Card: 12 Suit: 1 Color: 0   Card: 12 Suit: 3 Color: 1
```

Рис. 10.17. Результат выполнения программы, представленной на рис. 10.16

Совет по повышению эффективности 10.4

Хотя битовые поля позволяют сэкономить память, может случиться так, что полученный с их использованием машинный код будет исполняться заметно медленней. Причина этого кроется в том, что для доступа только к части адресуемой единицы памяти требуется применение дополнительных машинных операций. Это один из многочисленных примеров, когда необходимо делать выбор между быстродействием и компактностью.

10.11. Перечислимые константы

И наконец, язык C имеет еще один определяемый пользователем тип переменных, называемый *перечислением*. Перечисление объявляется при помощи ключевого слова **enum**, при этом задается имя переменной и определяется список именованных целых констант, называемый списком перечисления. Эти *перечислимые константы* фактически являются символическимими константами, значение которых может устанавливаться автоматически. Начальное значение в **enum** задается равным **0**, если не задано иначе, и увеличивается с шагом **1**. Например, перечисление

```
enum months {JAN, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP,
             OCT, NOV, DEC};
```

создает новый тип **enum months**, в котором идентификаторам автоматически присваиваются значения от **0** до **11**. Для того чтобы перенумеровать месяцы от 1 до 12, можно задать перечислимый тип следующим образом:

```
enum months {JAN = 1, FEB, MAR, APR, MAY, JUN, JUL, AUG,
             SEP, OCT, NOV, DEC};
```

Так как значение первого идентификатора явно устанавливается равным **1**, значение остальных увеличивается, начиная с **1**, и в результате мы получаем значения от **1** до **12**. Идентификаторы в перечислении должны иметь уникальное имя. Значения каждой перечислимой константы можно задавать явно путем присвоения значения идентификатору непосредственно в определении. Несколько элементов в списке перечисления могут иметь одно и тоже целое значение. В программе на рис. 10.18 перечислимая переменная **months** используется в структуре **for** для вывода на печать месяцев года из массива **monthName**. Отметим, что мы задали **monthName[0]** как пустую строку "". Некоторые программисты, возможно, предпочтут установить для **monthName[0]** значение *****ERROR*****, чтобы показать — произошла логическая ошибка.

```
/* Применение перечислимого типа */
#include <stdio.h>

enum months {JAN = 1, FEB, MAR, APR, MAY, JUN,
             JUL, AUG, SEP, OCT, NOV, DEC};

main()
{
   enum months month;
   char *monthName[] = {"", "January", "February", "March",
                        "April", "May", "June", "July",
                        "August", "September", "October",
                        "November", "December"};

   for (month = JAN; month <= DEC; month++)
      printf("%2d%11s\n", month, monthName[month]);

   return 0;
}
```

```
 1    January
 2   February
 3      March
 4      April
 5        May
 6       June
 7       July
 8     August
 9  September
10    October
11   November
12   December
```

Рис. 10.18. Использование перечисления

Распространенная ошибка программирования 10.17

Присвоение значения перечислимой константе после того, как она была определена, является синтаксической ошибкой.

Хороший стиль программирования 10.6

Для имен перечислимых констант рекомендуется использовать буквы только верхнего регистра. Это позволит выделить их в тексте программы и будет напоминать программисту, что перечислимые константы — это не переменные.

Резюме

- Структуры — это наборы (иногда называемые агрегатами) переменных, объединенных под одним именем.
- Структуры могут содержать переменные различных типов.
- Определение структуры начинается с ключевого слова **struct**. Между скобками определения структуры размещаются объявления ее элементов.
- Элементы в пределах структуры должны иметь уникальные имена.
- Определение структуры создает новый тип данных, который можно использовать для объявления переменных.
- Существует два способа объявления переменных структуры. Первый состоит в том, чтобы объявлять переменные так же, как это делается с переменными других типов данных, используя **struct tagName** как тип. Второй способ заключается в том, чтобы поместить переменные между закрывающей скобкой определения структуры и точкой с запятой, заканчивающей определение структуры.
- Имя-этикетка для структуры не обязательна. Если определение структуры не содержит имя-этикетку, переменные для этой структуры могут быть объявлены только в определении структуры, и никаких других новых переменных данного типа в дальнейшем объявить нельзя.
- Структуры можно инициализировать, используя список инициализации, для этого после имени переменной в объявлении структуры ставится знак равенства, за которым следует помещенный в фигурные скобки и разделенный запятыми список инициализаторов. Если инициализаторов в списке меньше, чем имеется элементов структуры, оставшимся элементам автоматически присваивается значение **0** (или **NULL**, если элемент — указатель).
- Одна структура может быть присвоена другой структуре того же типа.
- Допускается инициализация переменной-структуры с помощью переменной того же типа.
- Операция элемента структуры используется для обращения к элементу через имя переменной структуры.
- Операция указателя структуры, состоящая из знака минус (-) и знака больше (>) без пробела между ними, используется для обращения к элементу через указатель структуры.
- Если структура или отдельные ее элементы передаются функции, они передаются вызовом по значению.
- Чтобы осуществить вызов структуры по ссылке, необходимо передать адрес структуры.

- Массивы структур вызываются по ссылке автоматически.
- Чтобы вызвать массив по значению, создается структура, содержащая массив как элемент.
- Объявление нового имени с помощью **typedef** не создает нового типа, а создает лишь имя, являющееся синонимом для определенного ранее типа.
- Объединение — производный тип данных, элементы которого разделяют одну и ту же область памяти. Элементы объединения могут иметь произвольный тип.
- Память, зарезервированная под объединения, должна быть достаточной для хранения самого большого его элемента. В большинстве случаев объединение содержит два или более типа данных. В данный момент времени можно ссылаться только на один элемент и, соответственно, только один тип данных.
- Объединение объявляется посредством ключевого слова **union**. Формат объявления тот же, что и в случае структуры.
- При объявлении объединение можно инициализировать только значениями того же типа, что и его первый элемент.
- Поразрядная операция И (&) берет два целочисленных операнда и устанавливает бит результата равным 1 в том случае, если соответствующие биты в каждом из операндов равны 1.
- Маски используются, чтобы спрятать некоторые биты числа.
- Поразрядная операция включающего ИЛИ (|) берет два операнда и устанавливает бит результата равным 1 в том случае, если соответствующий бит хотя бы в одном из операндов равен 1.
- Для каждой из поразрядных операций (за исключением одноместной операции поразрядного дополнения) существует соответствующая операция присваивания.
- Поразрядная операция исключающего ИЛИ (^) берет два операнда и устанавливает бит результата равным 1 в том случае, если только один из соответствующих битов операндов равен 1.
- Операция сдвига влево (<<) смещает биты в левом операнде влево на число битов заданное правым операндом. Освобождаемые справа биты заполняются 0-ми.
- Операция сдвига вправо (>>) смещает биты в левом операнде вправо на число битов заданное правым операндом. При применении сдвига вправо к целому без знака освобождаемые слева биты заполняются 0-ми. В случае целого со знаком освобождаемые биты могут заполняться как 0-ми, так и 1-ми в зависимости от типа машины.
- Операция поразрядного дополнения (~) берет операнд и заменяет значения его битов на противоположные, при этом получается дополнение операнда до единицы.
- Битовые поля экономят память, храня данные в минимально необходимом числе бит.
- Элементы — битовые поля необходимо объявлять как **int** или **unsigned**.

- *Битовое поле объявляется с помощью двоеточия и ширины поля*, помещаемых после имен элементов типа **unsigned** или **int**.
- Ширина битового поля должна быть целой константой, значение которой заключено между 0 и числом бит, используемым для хранения переменной типа **int** в вашей системе.
- Битовое поле, заданное без имени, используется как заполнитель.
- Битовое поле без имени с шириной **0** используется для выравнивания следующего битового поля по границе нового машинного слова.
- Перечисление объявляется при помощи ключевого слова **enum**, при этом задается имя переменной и определяется список именованных целых констант, называемый списком перечисления. Значения в **enum** начинаются с **0**, если не задано иное число, и всегда увеличиваются на **1**.

Терминология

struct
typedef
union
агрегаты
битовое поле
битовое поле нулевой ширины
вложенные структуры
выбор между быстродействием и компактностью
дополнение
дополнение до единицы
доступ к элементам структуры
запись
заполнитель структуры
имя структуры
имя элемента
имя-этикетка
инициализация структур
маска
маскирование битов
массив структур
неименованное битовое поле
объявление структур
операция дополнения до единицы (~)
операция левого сдвига (<)
операция поразрядного И (&)
операция поразрядного включающего ИЛИ (|)
операция поразрядного исключающего ИЛИ (^)
операция правого сдвига (>)
операция присваивания левого сдвига (<=)
операция присваивания поразрядного И (&=)
операция присваивания поразрядного включающего ИЛИ (|=)
операция присваивания поразрядного исключающего ИЛИ (^=)
операция присваивания правого сдвига (>=)
операция элемента структуры — точка (.)
операция элемента структуры — стрелка (->)
определение структур
перечисление
перечислимая константа
поразрядные операции
присваивание структур
производный тип
сдвиг
сдвиг влево
сдвиг вправо
структура, ссылающаяся на себя
тип структуры
типы данных, определяемые программистом
указатель на структуру
ширина битового поля
элемент
этикетка структуры

Распространенные ошибки программирования

10.1. Забывают о точке с запятой, завершающей определение структуры.

10.2. Присваивают структуру одного типа структуре другого типа.

10.3. Сравнение структур является синтаксической ошибкой, потому что на различных системах существуют отличающиеся друг от друга правила выравнивания.

10.4. Вставляют пробел между составными частями — и > операции указателя структуры (или вставляют пробелы между составными частями других вводимых при помощи нескольких символов операций, за исключением ?:).

10.5. Пытаются сослаться на элемент структуры, используя только имя элемента.

10.6. Не используют скобки, когда ссылаются на элемент структуры, используя указатель и операцию элемента структуры (иными словами, запись ***aPtr.suit** является синтаксической ошибкой).

10.7. Полагая, что структуры, как и массивы, вызываются по ссылке автоматически, пытаются изменить значение структуры вызывающей функции в вызываемой.

10.8. Забывают индекс массива при ссылке на отдельные структуры в массиве структур.

10.9. Обращаются к данным, хранящимся в объединении, как к данным одного типа, в то время как, хранящиеся в данный момент в объединении данные принадлежат другому типу, что приводит к появлению логической ошибки.

10.10. Сравнение объединений является синтаксической ошибкой, поскольку в разных системах существуют различные требования на выравнивание.

10.11. Инициализация объединения в объявлении значением, которое имеет тип, отличный от типа первого элемента объединения.

10.12. Использование логической операции И (&&) вместо поразрядной операции (&) и наоборот.

10.13. Использование логической операции ИЛИ (||) вместо поразрядной операции (|) и наоборот.

10.14. Значение, получаемое в результате сдвига, становится неопределенным, если правый операнд имеет отрицательное значение или его значение больше, чем число бит, в которых хранится левый операнд.

10.15. Попытка обратиться к отдельным битам битового поля так, как если бы они были элементами массива. Битовые — поля это вовсе не «массивы бит».

10.16. Попытка взять адрес битового поля (операция & неприменима к битовым полям, так как они не имеют адреса).

10.17. Присвоение значения перечислимой константе, после того как она была определена, является синтаксической ошибкой.

Хороший стиль программирования

10.1. Давать каждой создаваемой структуре имя-этикетку. Имя-этикетка структуры позволит объявлять далее в программе новые переменные, принадлежащие к типу этой структуры.

10.2. Осмысленный выбор имени-этикетки структуры позволит сделать программу самодокументированной.

10.3. Рекомендуется избегать использования одинаковых имен для элементов структур различного типа. Это не запрещено, но может привести к недоразумениям.

10.4. Не рекомендуется вставлять пробелы слева и справа от знаков операций. и ->. Это поможет подчеркнуть, что выражение, в которых содержаться данные операции, по существу являются именем одной переменой.

10.5. Начинать имена, определяемые **typedef**, с прописной буквы, чтобы подчеркнуть тот факт, что эти имена являются синонимами имен другого типа.

10.6. Для имен перечислимых констант рекомендуется использовать буквы только верхнего регистра. Это позволит выделить их в тексте программы и будет напоминать программисту, что перечислимые константы — это не переменные.

Советы по переносимости программ

10.1. На разных машинах элементы данных отличаются размером, а кроме того, от типа машины зависит организация выравнивания памяти, поэтому и расположение структуры в памяти также зависит от типа машины.

10.2. Использование **typedef** помогает облегчить адаптацию программы к различным аппаратно-программным платформам.

10.3. Если сохранять данные в объединении, как принадлежащие одному типу, а обращаться к ним в дальнейшем как к данным другого типа, то результат будет зависеть от реализации системы.

10.4. Объем памяти необходимый для хранения объединения зависит от типа компьютера.

10.5. Некоторые объединения невозможно просто перенести на другую компьютерную систему. Является ли объединение переносимым или нет, часто зависит от требований на выравнивание памяти для типов данных элементов объединения в данной системе.

10.6. Поразрядные операции с битовыми данными являются машинно-зависимыми.

10.7. Реализация операции сдвиг вправо зависит от типа машины. Сдвиг вправо для отрицательного целого со знаком заполняет освобождаемые биты **0**-ми на одних машинах и **1**-ми на других.

10.8. Операции с битовыми полями зависят от типа машины. Некоторые компьютеры, например, позволяют битовым полям пересекать границы слов, в то время как другие нет.

Советы по повышению эффективности

10.1. Вызов структуры по ссылке более эффективен, чем вызов по значению (при котором происходит копирование всей структуры целиком).

10.2. Объединения позволяют сэкономить память.

10.3. Битовые поля экономят память.

10.4. Хотя битовые поля позволяют сэкономить память, может получиться так, что полученный с их использованием машинный код будет исполняться заметно медленней. Причина этого кроется в том, что для доступа только к части адресуемой единицы памяти требуется применение дополнительных машинных операций. Это один из многочисленных примеров, когда необходимо делать выбор между быстродействием и компактностью.

Общие методические замечания

10.1. Как и в случае объявления **struct**, объявление **union** просто создает новый тип. Размещение объявления **union** или **struct** вне функции не создает глобальной переменной.

Упражнения для самоконтроля

10.1. Заполните пропуски в каждом из следующих утверждений.

a) _____ — это объединение логически связанных переменных под одним именем.

b) _____ — это объединение переменных под одним именем, в котором переменные разделяют одну и ту же область памяти.

c) В результате выполнения операции _____ каждый бит значения результата устанавливается равным 1, если соответствующие биты каждого операнда равны 1. В противном случае бит устанавливаются равным 0.

d) Переменные, объявленные в определении структуры, называются ее _____.

e) В результате выполнения операции _____ каждый бит значения результата устанавливается равным 1, если хотя бы один из соответствующих битов любого из операндов равен 1. В противном случае бит устанавливается равным 0.

f) Ключевое слово _____ объявляет структуру.

g) Ключевое слово _____ используется для задания синонимов ранее определенных типов данных.

h) Биты в результате выполнения операции _____ устанавливаются равными 1, если ровно один соответствующий бит любого из операндов равен 1. В противном случае биты устанавливаются равными 0.

i) Операция поразрядного AND (&) часто используется для того, чтобы _____ отдельные биты, то есть выделить некоторые из битов, «обнулив» остальные.

j) Ключевое слово _____ используется, чтобы ввести определение объединения.

k) Имя структуры называют _____ структуры.

l) Обращение к элементу структуры возможно либо с помощью операции _____, либо с помощью операции _____.

m) Операции _____ и _____ применяются для сдвига битов влево или вправо соответственно.

n) _____ является набором целых чисел, представленных идентификаторами.

10.2. Установите, являются следующие утверждения верными или неверными; если утверждение неверно, объясните, почему.

a) Структуры могут содержать только один тип данных.

b) Два объединения можно сравнить между собой, чтобы определить, равны ли они между собой.

c) Наличие имени-этикетки у структуры является необязательным.

d) Элементы различных структур должны иметь уникальные имена.

e) Ключевое слово **typedef** используется для определения новых типов данных.

f) Структуры всегда передаются функциям по ссылке.

g) Структуры нельзя сравнивать.

10.3. Напишите один или несколько операторов C, выполняющих каждое из следующих действий:

a) Определите структуру с названием **part**, содержащую переменную **partNumber** типа **int** и массив **partName** типа **char**, значения элементов которого могут иметь длину до 25 символов.

b) Определите, что **Part** является синонимом типа **struct part**.

c) Используйте **Part** для объявления переменной **a** типа **struct part**, массива **b[10]** типа **struct part** и переменной **ptr** — указателя на **struct part**.

d) Считайте с клавиатуры **partNumber** и **partName** в отдельные элементы переменной **a**.

e) Присвойте значение переменной **a** третьему элементу массива **b**.

f) Присвойте адрес массива **b** указателю **ptr**.

g) Выведите на печать значение элемента 3 массива **b**, используя переменную **ptr** и операцию указателя структуры для обращения к ее элементам.

10.4. Найдите ошибки в каждом из следующих примеров:

a) Предположим, что **struct card** была определена как содержащая два указателя на тип **char**, а именно **face** и **suit**. Кроме того, была объявлена переменная **c** типа **struct card** и переменная **cPtr** как

указатель на **struct card**. Переменной **cPtr** был присвоен адрес переменной **c**.

```
printf("%s\n", *cPtr->face);
```

b) Предположим, было определено, что **struct card** содержит два указателя типа **char**, а именно **face** и **suit**. Кроме того, было объявлено, что массив **hearts[13]** принадлежит к типу **struct card**. Следующий оператор должен вывести на печать **face** для десятого элемента.

```
printf("%s\n", hearts.face);
```

c)
```
union values {
   char w;
   float x;
   double y;
} v = {1.27};
```

d)
```
struct person {
   char lastName[15];
   char firstName[15];
   int age;
}
```

e) Предположим, что **struct person** была определена как в примере (d), разумеется, с соответствующими исправлениями.

```
person d;
```

f) Предположим, было объявлено, что переменная **p** принадлежит к типу **struct person**, и переменная **c** объявлена принадлежащей к типу **struct card**.

```
p = c;
```

Ответы на упражнения для самопроверки

10.1. a) структура, b) объединение, c) поразрядного И (&), d) элементами, e) поразрядного включающего ИЛИ (|), f) **struct**, g) **typedef**, h) поразрядного включающего ИЛИ (^), i) маскировать, j) **union**, k) этикеткой, l) элемента структуры, указателя структуры, m) сдвига влево(<<), сдвига вправо(>>), n) список перечисления.

10.2. a) Неверно. Структура может содержать несколько типов данных.

b) Неверно. Объединения не могут сравниваться, по тем же причинам, что и структуры, из-за проблем связанных с выравниванием.

c) Верно.

d) Неверно. Элементы разных структур могут иметь одинаковые имена, но элементы одной структуры должны иметь уникальные имена.

e) Неверно. Ключевое слово **typedef** используется для определения новых имен (синонимов) для ранее определенных типов данных.

f) Неверно. Структуры всегда передаются функциям по значению.

g) Верно, из-за проблем, связанных с выравниванием.

10.3. a)
```
struct part {
      int partNumber;
      char partName[25];
};
```

b)
```
typedef struct part Part;
```

c)
```
Part a, b[10], *ptr;
```

d)
```
scanf("%d%s", &a.partNumber, &a.partName);
```

e)
```
b[3] = a;
```

f)
```
ptr = b;
```

g)
```
printf("%d %s\n", (ptr + 3)->partNumber,
   (ptr + 3)->partName);
```

10.4. a) Ошибка: пропущены скобки, в которые необходимо заключить ***cPtr**, что приведет к неправильному порядку операций при вычислении выражения.

b) Ошибка: был пропущен индекс массива. Выражение следует записать следующим образом **hearts[10].face**.

c) Ошибка: объединение может быть инициализировано только значениями, которые имеют тот же самый тип, что и первый элемент объединения.

d) Ошибка: в конце определения структуры должна стоять точка с запятой.

e) Ошибка: в объявлении переменной было пропущено ключевое слово **struct**.

f) Ошибка: нельзя присваивать переменные-структуры различных типов друг другу.

Упражнения

10.5. Напишите определения для следующих структур и объединений:

a) Структуры **inventory**, содержащей следующие элементы: символьный массив **partName[30]**, целую переменную **partNumber**, действительную переменную **price**, целую переменную **stock** и целую переменную **reorder**.

b) Объединения **data**, содержащего **char c**, **short s**, **long l**, **float f**, **double d**.

c) Структуры с именем **address**, содержащей символьные массивы **streetAddress[25]**, **city[20]**, **state[3]** и **zipCode[6]**.

d) Структуры **student**, содержащей массивы **firstName[15]** и **lastName[15]**, а также переменную **homeAddress** типа **struct address** из вопроса (c).

e) Структуры **test**, содержащей 16 битовых полей шириной в один бит. В качестве имен битовых полей возьмите буквы от **a** до **p**.

10.6. Задано определение структуры и объявлены переменные:

```
struct customer {
   char lastName[15];
   char firstName[15];
   int customerNumber;
```

```
   struct {
      char phoneNumber[11];
      char address[50];
      char city[15];
      char state[3];
      char zipCode[6];
   } personal;

} customerRecord, *customerPtr;

customerPtr = &customerRecord;
```

Напишите отдельные выражения, которые можно использовать для обращения к элементам структуры в каждом из следующих случаев.

a) Элементу **lastName** структуры **customerRecord**.

b) Элементу **lastName** структуры, на которую указывает **customerPtr**.

c) Элементу **firstName** структуры **customerRecord**.

d) Элементу **firstName** структуры, на которую указывает **customerPtr**.

e) Элементу **customerNumber** структуры **customerRecord**.

f) Элементу **customerNumber** структуры, на которую указывает **customerPtr**.

g) Элементу **phoneNumber** элемента **personal** структуры **customerRecord**.

h) Элементу **phoneNumber** элемента **personal** структуры, на которую указывает **customerPtr**.

i) Элементу **address** элемента **personal** структуры **customerRecord**.

j) Элементу **address** элемента **personal** структуры, на которую указывает **customerPtr**.

k) Элементу **city** элемента **personal** структуры **customerRecord**.

l) Элементу **city** элемента **personal** структуры, на которую указывает **customerPtr**.

m) Элементу **state** элемента **personal** структуры **customerRecord**.

n) Элементу **state** элемента **personal** структуры, на которую указывает **customerPtr**.

o) Элементу **zipCode** элемента **personal** структуры **customerRecord**.

p) Элементу **zipCode** элемента **personal** структуры, на которую указывает **customerPtr**.

10.7. Измените программу на рис. 10.16 так, чтобы колода тасовалась, применив эффективный алгоритм тасования (см. рис. 10.3). Распечатайте полученную колоду в две колонки как на рис. 10.4. Перед каждой картой должен стоять ее цвет.

10.8. Создайте объединение **integer** с элементами **char c**, **short s**, **int i** и **long l**. Напишите программу, которая вводит значения типов **char**, **short**, **int**, **long**, и сохраните значения в переменных типа **union integer**. Каждая переменная-объединение должна быть напечатана

как **char**, **short**, **int** и **long**. Всегда ли значения печатаются правильно?

10.9. Создайте объединение **floatingPoint** с элементами **float f**, **double d** и **long double l**. Напишите программу, которая вводит значения типов **float**, **double** и **long double**. Сохраните значения в переменных типа **union floatingPoint**. Каждая переменная-объединение должна быть напечатана как **float**, **double** и **long double**. Всегда ли значения печатаются правильно?

10.10. Напишите программу, которая сдвигает целое число вправо на четыре бита. Программа должна распечатывать целое число в двоичном представлении до и после операции сдвига. Выясните, нули или единицы помещает ваша система в освободившиеся биты.

10.11. Если ваш компьютер имеет четырехбайтовый тип **int**, модернизируйте программу на рис. 10.7 так, чтобы она работала с четырехбайтовыми целыми.

10.12. Сдвиг влево целого числа типа **unsigned** на один бит эквивалентен умножению на два. Напишите функцию **power2**, которая берет два целых аргумента **number** и **pow** и вычисляет

$$number * 2^{pow}$$

Используйте операцию сдвига. Программа должна выдавать на печать число как в десятичном, так и в двоичном виде.

10.13. Операция сдвига влево может применяться для того, чтобы упаковать два символьных значения в двухбайтовое целое без знака. Напишите программу, которая принимает два символа с клавиатуры и передает их функции **packCharacters**. Чтобы упаковать два символа в целую переменную без знака, присвойте первый символ переменной типа **unsigned**, сдвиньте переменную влево на 8 двоичных позиций и объедините ее со вторым символом посредством операции поразрядного включающего ИЛИ. Программа должна выводить символы в двоичном формате до и после того, как они будут упакованы в целое без знака, чтобы доказать, что символы действительно правильно упакованы в переменную.

10.14. Используя операцию сдвига вправо, операцию поразрядного И и маску, напишите функцию **unpackCharacters**, которая берет целое типа **unsigned** из упражнения 10.13 и распаковывает его в два символа. Для того чтобы распаковать два символа из двухбайтового целого без знака, объедините по И беззнаковое целое с маской **65280 (11111111 00000000)** и сдвиньте результат вправо на восемь бит. Присвойте полученное значение переменной типа **char**. После этого объедините по И беззнаковое целое с маской **255 (00000000 11111111)**. Присвойте полученное значение другой переменной типа **char**. Программа должна напечатать беззнаковое целое в двоичном виде, прежде чем его распаковать, а потом напечатать символы в двоичном виде, чтобы подтвердить, что они были распакованы правильно.

10.15. Если ваша система имеет 4-байтовые целые, перепишите программу упражнения 10.13 для упаковывания 4-х символов.

10.16. Если ваша система имеет 4-байтовые целые, перепишите функцию **unpackCharacters** упражнения 10.14 для распаковывания четырех символов. Создавая маски, которые понадобятся для распаковки четырех символов, посредством сдвига значения 255 в переменной маски на 8 бит 0, 1, 2 или 3 раза (в зависимости от байта, который вы извлекаете).

10.17. Напишите программу, которая меняет порядок битов в целом числе без знака на обратный. Программа должна принимать введенное пользователем значение и вызывать функцию **reverseBits** для того, чтобы вывести на печать биты в обратном порядке. Выведите значение в двоичном виде как до, так и после преобразования, чтобы подтвердить, что биты действительно располагаются в обратном порядке.

10.18. Измените функцию **displayBits** на рис. 10.7 так, чтобы ее можно было использовать как на системах с 2-байтовыми, так и с 4-байтовыми целыми. Подсказка: чтобы определить, сколько байт выделяется под целое число на конкретной машине, используйте операцию **sizeof**.

10.19. Нижеследующая программа вызывает функцию **multiple**, чтобы определить, является ли целое число, введенное с клавиатуры, кратным некоторому целому числу **X**. Исследуйте функцию **multiple** и скажите, чему равно значение **X**.

```
/* Программа определяет, является ли целое число кратным x */
#include <stdio.h>

int multiple(int);
main()
{
   int y;

   print("Enter an integer between 1 and 32000; ");
   scanf("%d", &y);

   if (multiple(y))
      printf("%d is a multiple of X\n", y);
   else
      printf("%d is not a multiple of X\n", y);

   return 0;
}

int multiple(int num)
{
   int i, mask = 1, mult = 1;

   for (i = 1; i <= 10; i++, mask <<= 1)
      if ((num & mask) != 0) {
         mult = 0;
         break;
      }

   return mult;
}
```

10.20. Что делает следующая программа?

```
#include <stdio.h>

int mystery(unsigned);

main()
{
   unsigned x;

   printf("Enter an integer: ");
   scanf("%u", &x);
   printf("The result is %d\n", mystery(x));
   return 0;
}

int mystery(unsigned bits)
{
   unsigned i, mask = 1 << 15, total = 0;

   for (i = 1; i <= 16; i++, bits <<= 1)
      if ((bits & mask) == mask)
         ++total;

return total % 2 == 0 ? 1 : 0;
}
```

глава

11

Работа с файлами



Цели

- Научиться создавать, читать, записывать и модифицировать файлы.
- Познакомиться с обработкой файлов последовательного доступа.
- Познакомиться с обработкой файлов произвольного доступа.

Содержание

11.1. Введение
11.2. Иерархия данных
11.3. Файлы и потоки
11.4. Создание файла последовательного доступа
11.5. Чтение данных из файла последовательного доступа
11.6. Файлы произвольного доступа
11.7. Создание файлов произвольного доступа
11.8. Произвольная запись данных в файл произвольного доступа
11.9. Последовательное чтение данных из файла произвольного доступа
11.10. Пример: программа обработки транзакций

Резюме • Распространенные ошибки программирования • Хороший стиль программирования • Советы по переносимости программ • Советы по повышению эффективности • Упражнения для самоконтроля • Ответы на упражнения для самоконтроля • Упражнения

11.1. Введение

Хранение данных в переменных и массивах является временным; все эти данные теряются при завершении работы программы. Для постоянного хранения больших объемов данных используются *файлы*. Компьютеры хранят данные на устройствах вторичной памяти, главным образом дисковых устройствах. В этой главе мы объясним, как с помощью написанных на C программ создавать, обновлять и обрабатывать файлы данных. Будут рассмотрены файлы с последовательным и произвольным доступом к данным.

11.2. Иерархия данных

Компьютер обрабатывает элементы данных в двоичном виде, то есть в *виде комбинаций нулей и единиц*. Дело в том, что проще и экономичнее изготовлять электронные схемы, которые могут находиться в двух устойчивых состояниях — одно из таких состояний представляет **0**, а другое **1**. Удивительно, но все впечатляющие задачи, которые способен выполнять компьютер, основываются на самых элементарных манипуляциях с нулями и единицами.

Значение наименьшего элемента данных, к которому может обращаться компьютер, равно **0** или **1**. Этот элемент данных называется *бит* (по-английски *bit* сокращение от «*bi*nary digi*t*» — число, способное принимать одно из двух значений). Совокупность операций, выполняемых компьютером, сводится к различным простым действиям над битами, таким, как определение значения бита, установка значения бита, инверсия бита (**1** вместо **0** и наоборот).

Для программистов очень обременительно работать с данными низкого уровня, какими являются биты. Вместо этого программисты предпочитают работать с данными в виде *десятичных цифр* (то есть 0, 1, 2, 3, 4, 5, 6, 7, 8 и 9), *букв* (от A до Z и от a до z), и *специальных знаков* ($, @, %, *, (,), -, +, ", :, ?, / и многих других). Цифры, буквы и специальные знаки называются *символами*. Множество символов, которыми можно пользоваться при написании программ и представлении элементов данных на конкретном компьютере, называется его *набором символов*. Так как компьютеры могут обрабатывать только нули и единицы, каждый символ из набора символов компьютера представляется комбинацией из **0** и **1** (называемой *байтом*). На сегодняшний день байты чаще всего состоят из восьми битов. Программисты записывают программы и элементы данных, пользуясь символами; компьютеры в дальнейшем обрабатывают эти символы как наборы битов.

Подобно тому, как символы состоят из битов, *поля* состоят из символов. Поле представляет собой группу символов, которые передают значение. Например, поле, состоящее исключительно из букв верхнего и нижнего регистра, может использоваться для представления имени человека.

Обрабатываемые компьютерами элементы образуют *иерархию данных*, в которой элементы данных становятся больше по размеру и сложнее по структуре по мере продвижения от битов к символам (байтам), полям и т.д.

Запись (соответствует в C понятию **struct**) состоит из нескольких полей. Например, в программе для начисления заработной платы запись для конкретного работника может состоять из следующих полей:

1. Номер карточки социального страхования

2. Имя

3. Адрес

4. Ставка почасовой оплаты

5. Количество заявленных налоговых льгот

6. Выплаты на данный момент

7. Размер удержанного федерального налога, и т.д.

Таким образом, запись — это группа логически связанных полей. В предыдущем примере каждое поле принадлежало одному и тому же работнику. Конечно, компания может иметь много работников и будет иметь запись о начислении заработной платы для каждого из них. *Файлом* называется группа связанных записей. Файл начисления заработной платы содержит по одной записи на каждого работника. Таким образом, файл начисления заработной платы маленькой компании может содержать 22 записи, в то время как такой же файл большой компании может содержать 100000 записей. Вполне обычна ситуация, когда организация имеет сотни и даже тысячи файлов, которые могут содержать миллионы или даже миллиарды символов информации. С рос-

том популярности лазерных компакт-дисков и технологии мультимедиа даже триллионы байт скоро станут обычным явлением. Рис. 11.1 иллюстрирует иерархию данных.

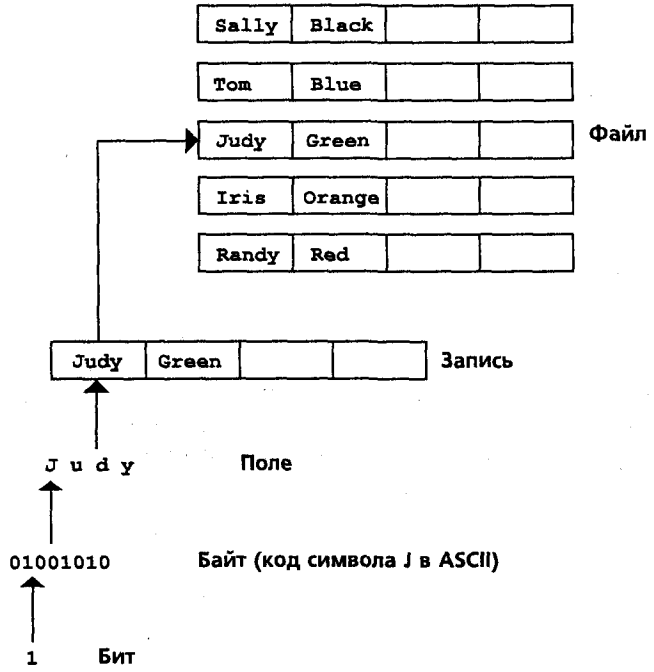


Рис. 11.1. Иерархия данных

Чтобы облегчить поиск необходимых данных, по крайней мере одно поле в каждой записи файла выбирается в качестве *ключа записи*. Ключ идентифицирует запись как принадлежащую конкретному лицу или объекту. Например, в записи начисления заработной платы, описанной в этом разделе, в качестве ключа удобно выбрать номер карточки социального страхования.

Существует множество способов организации записей в файле. Наиболее популярный из них называется *последовательным файлом*, в котором записи, как правило, хранятся в порядке, определяемом ключом. В файле начисления заработной платы, например, записи хранились бы упорядоченными по номеру карточки социального страхования.

Большинство фирм используют для хранения данных много различных файлов. Например, компании могут иметь файлы начисления заработной платы, файлы счетов, по которым необходимо получить деньги (списки сумм, причитающихся с клиентов), файлы инвентаризации, а также многочисленные файлы других типов. Группа связанных файлов иногда называется *базой данных*. Набор программ, предназначенный для создания и управления базами данных, называется *системой управления базами данных* (СУБД, или DBMS — Database Management System).

11.3. Файлы и потоки

С рассматривает любой файл как последовательный поток байтов (рис. 11.2). Каждый файл оканчивается или *маркером конца файла*, или особым байтом, определенным в работающей с файлами программе. Когда файл *открывается*, ему ставится в соответствие поток. В начале исполнения программы автоматически открываются три файла и связанные с ними потоки — *стандартный ввод*, *стандартный вывод* и *стандартная ошибка*. Потоки обеспечивают каналы передачи данных между файлами и программами. Например, стандартный поток ввода позволяет программе считывать данные с клавиатуры, а стандартный поток вывода позволяет выводить данные на экран. Открытый файл возвращает указатель на структуру **FILE** (определенную в **<stdio.h>**), которая содержит информацию, используемую при работе с файлом. Эта структура включает *дескриптор файла*, то есть индекс в массиве операционной системы, называемом *таблицей открытых файлов*. Каждый элемент массива содержит *блок управления файлом* (FCB), который используется операционной системой для доступа к конкретному файлу. Для обращения к стандартному вводу, стандартному выводу и стандартному потоку ошибок следует воспользоваться указателями файлов **stdin**, **stdout** и **stderr** соответственно.

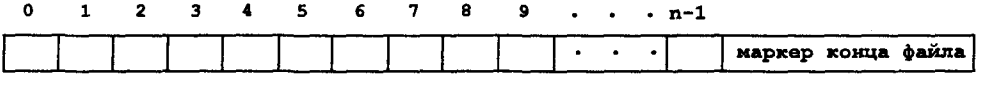


Рис. 11.2. Вид файла из n байт с точки зрения C

Стандартная библиотека поддерживает многочисленные функции чтения данных из файлов и записи данных в файлы. Функция **fgetc**, подобно **getchar**, считывает из файла один символ. Функция **fgetc** получает в качестве аргумента указатель на **FILE** для файла, из которого будет считываться символ. Вызов **fgetc(stdin)** считывает один символ из **stdin** — стандартного ввода. Такой ее вызов эквивалентен вызову **getchar()**. Функция **fputc**, подобно **putchar**, записывает один символ в файл. Функция получает в качестве аргумента символ, который должен быть записан. Вызов функции **fputc('a',stdout)** записывает символ **'a'** в **stdout** — стандартный вывод. Такой вызов этой функции эквивалентен **putchar('a')**.

Несколько других функций, использовавшихся для чтения данных из стандартного ввода и записи в стандартный вывод, имеют аналоги со сходными названиями, предназначенные для выполнения операций с файлами. Например, функции **fgets** и **fputs** могут быть использованы соответственно для чтения строки из файла и записи строки в файл. Их «двойники» **gets** и **puts** обсуждались в главе 8. В следующих разделах мы представим аналоги функций **scanf** и **printf** — **fscanf** и **fprintf**. А несколько позже обсудим функции **fread** и **fwrite**.

11.4. Создание файла последовательного доступа

В C не предусмотрено возможности задания структуры файла. Другими словами, не существует никаких операций с записями в файле, реализованных как часть языка C. Поэтому программист должен сам позаботиться о создании структуры файла, отвечающей требованиям конкретного приложения.

В следующем примере мы увидим, как программист может задать подобную структуру записи.

Программа на рис. 11.3 создает простой файл последовательного доступа, который можно использовать в программе учета оплаты счетов, помогающей следить за суммами задолженности клиентов компании. Для каждого клиента программа просит ввести номер счета, имя клиента и баланс (то есть сумму, которую клиент должен компании за товары и услуги, полученные в прошлом). Данные на каждого из клиентов образуют «запись» для этого клиента. В этом приложении в качестве ключа записи используется номер счета. Другими словами, файл будет создаваться и поддерживаться упорядоченным по номерам счетов. Эта программа подразумевает, что пользователи вводят записи в порядке возрастания номеров. В более удобной для работы системе регистрации счетов должна обеспечиваться возможность сортировки, чтобы пользователь мог вводить записи в произвольном порядке. В этом случае записи должны сначала упорядочиваться, а затем уже записываться в файл.

```
/* Создание последовательного файла */
#include <stdio.h>

main()
{
   int account;
   char name[30];
   float balance;
   FILE *cfPtr;   /*cfPtr = указатель файла clients.dat */

   if ((cfPtr =  fopen("clients.dat", "w")) == NULL)
      printf("File could not be opened\n");
   else {
      printf("Enter the account, name, and balance.\n");
      printf("Enter EOF to end input.\n");
      printf("? ");
      scanf("%d%s%f", &account, name, &balance);

      while (!feof(stdin)) {
         fprintf(cfPtr, "%d %s %.2f\n",
                 account, name, balance);
         printf("? ");
         scanf("%d%s%f", &account, name, &balance);
      }

      fclose(cfPtr);
   }
   return 0;
}
```

```
Enter the account, name, and balance.
Enter EOF to end input.
? 100 Jones 24.98
? 200 Doe 345.67
? 300 White 0.00
? 400 Stone -42.16
? 500 Rich 224.62
?
```

Рис. 11.3. Создание файла последовательного доступа

Теперь давайте посмотрим, как устроена эта программа. Оператор

```
FILE *cfPtr;
```

объявляет, что **cfPtr** является указателем на структуру **FILE**. Программа на языке С управляет каждым файлом при помощи отдельной структуры **FILE**. Чтобы работать с файлами, программисту нет необходимости знать, как устроена эта структура. Скоро мы увидим, как структура **FILE** косвенно ссылается на блок управления файлом (FCB) операционной системы для заданного файла.

Совет по переносимости программ 11.1

Структура **FILE** зависит от типа операционной системы (т.е. элементы ее меняются в зависимости от того, как каждая система обращается со своими файлами).

Каждый открытый файл должен иметь отдельно объявленный указатель типа **FILE**, который используется для ссылок на файл. Строка

```
if ((cfPtr = fopen("clients.dat", "w")) == NULL)
```

указывает имя файла — **"clients.dat"**, — который будет использоваться программой, и устанавливает «канал общения» с файлом. Для файла, открытого **fopen**, указателю файла **cfPtr** присваивается указатель на структуру **FILE**. Функции **fopen** передаются два аргумента: имя файла и *режим открытия файла*. Режим **"w"** указывает, что файл должен быть открыт для *записи*. Если файла не существует, и его открывают для записи, **fopen** создает файл. Если для записи открывается уже существующий файл, его содержимое уничтожается без предупреждения. Структура **if** используется в программе для того, чтобы определить, не является ли указатель файла **cfPtr** равным **NULL** (т. е. файл не открыт). Если он равен **NULL**, печатается сообщение об ошибке и выполнение программы заканчивается. В противном случае вводимые данные обрабатываются и записываются в файл.

Распространенная ошибка программирования 11.1

Открытие ранее созданного файла для записи (**"w"**), тогда как на самом деле пользователь хочет сохранить находящиеся в файле данные; содержимое файла уничтожается без предупреждения.

Распространенная ошибка программирования 11.2

Забывают открыть файл, перед тем как ссылаться на него в программе.

Программа предлагает пользователю ввести различные поля для каждой записи либо комбинацию, представляющую конец файла, когда ввод данных окончен. На рис. 11.4 представлен список комбинаций клавиш, означающих конец файла в различных компьютерных системах.

Строка

```
while (!feof(stdin))
```

использует функцию **feof**, чтобы определить, установлен ли *индикатор конца файла*, на который ссылается **stdin**. Индикатор конца файла сообщает программе, что чтение данных закончено. В программе на рис. 11.3 индикатор конца файла для стандартного ввода устанавливается, когда пользователь вводит соответствующую комбинацию клавиш. Аргумент, передаваемый функ-

ции **feof**, — указатель на файл, для которого необходимо проверить индикатор конца файла (в данном случае **stdin**). Функция возвращает ненулевое значение (true), если индикатор конца файла установлен, в противном случае возвращается нуль. В данной программе цикл **while**, включающий в себя вызов **feof**, будет продолжать выполняться, пока не будет установлен индикатор конца файла.

Компьютерные системы	Комбинация клавиш
UNIX	*<return> <ctrl>d*
IBM PC и совместимые	*<ctrl>z*
Macintosh	*<ctrl>d*
VAX(VMS)	*<ctrl>z*

Рис. 11.4. Комбинации клавиш для конца файла в различных популярных системах

Оператор

```
fprintf(cfPtr, "%d %s %.2f\n", account, name, balance);
```

записывает данные в файл **clients.dat**. Позже эти данные могут быть считаны программой, предназначенной для чтения файлов (см. раздел 11.5). Функция **fprintf** — это аналог **printf**, за исключением того, что **fprintf** получает еще аргумент — указатель файла для файла, в который будут записаны данные.

Распространенная ошибка программирования 11.3

Ссылаясь на файл, используют неверный указатель файла.

Хороший стиль программирования 11.1

Убедитесь, что вызовы функций, работающих с файлами, содержат корректные указатели файлов.

После того, как пользователь ввел конец файла, программа закрывает файл **clients.dat** с помощью **fclose** и завершает работу. Функции **fclose** в качестве аргумента также передается указатель файла (а не имя файла). Если функция **fclose** не вызывается явно, операционная система обычно сама закрывает файлы при завершении работы программы. Это пример поддержания операционной системой «порядка в доме».

Хороший стиль программирования 11.2

Явно закрывать каждый файл, как только выяснится, что программа больше не будет обращаться к файлу.

Совет по повышению эффективности 11.1

Закрывая файл, мы освобождаем ресурсы, которые, возможно, требуются другим пользователям или программам.

На рис. 11.3 представлен образец работы программы; пользователь вводит информацию для пяти счетов, а затем конец файла, чтобы показать окончание ввода данных. Здесь мы не имеем возможности убедиться, что записи данных в действительности появляются в файле. Для проверки того, что файл был успешно создан, в следующем разделе будет представлена программа, которая считывает содержимое файла и выводит его на печать.

Рис. 11.5 иллюстрирует связь между указателями на **FILE**, структурами **FILE** и FCB в памяти компьютера. Когда файл "**clients.dat**" открыт, FCB для файла скопирован в память. Рисунок показывает связь между возвращаемым указателем файла и FCB, используемым операционной системой для управления файлом.

Программы могут либо вообще не работать с файлами, либо работать с одним или несколькими файлами. Каждый из обрабатываемых в программе файлов должен иметь уникальное имя и будет иметь отличающийся от других указатель файла, возвращаемый **fopen**. Все функции работы с файлами последова-

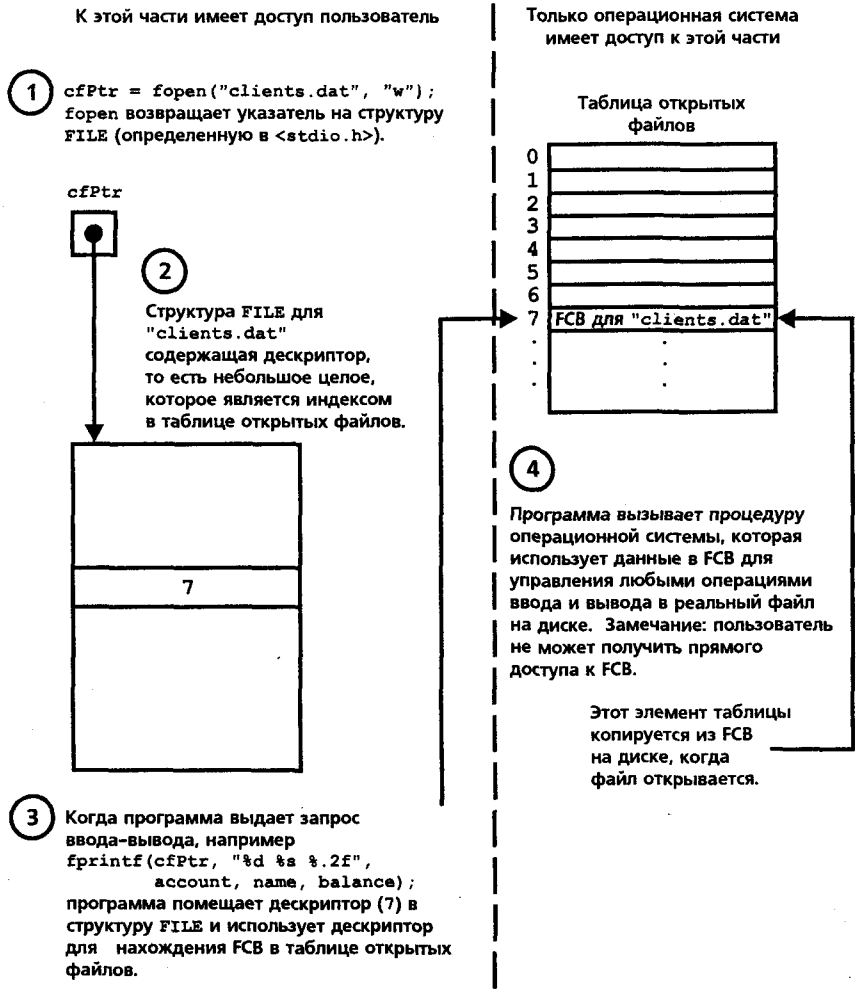


Рис. 11.5. Связь между указателями **FILE**, структурами **FILE** и FCB

тельного доступа после того, как файл открыт, должны ссылаться на файл с помощью соответствующего указателя файла. Файлы могут быть открыты в одном из нескольких возможных режимов. Чтобы создать файл или уничтожить содержимое файла, в который ранее были записаны данные, откройте файл для записи ("w"). Чтобы прочитать существующий файл, откройте файл для чтения ("r"). Добавить записи в конец уже существующего файла можно, открыв файл для добавления ("a"). Если необходимо как записывать в файл, так и считывать из него данные, откройте файл для обновления в одном из трех предназначенных для этого режимов — "r+", "w+" или "a+". Режим "r+" открывает файл для чтения и записи. Режим "w+" создает файл для чтения и записи. Если файл уже существует, файл открывается и текущее содержание файла уничтожается. Режим "a+" открывает файл для чтения и записи, все записи производятся в конец файла. Если файл не существует, он создается.

Если при открытии файла, вне зависимости от выбранного режима, происходит ошибка, **fopen** возвращает **NULL**. Некоторые часто встречающиеся ошибки перечислены ниже:

Распространенная ошибка программирования 11.4

Открытие для чтения несуществующего файла.

Распространенная ошибка программирования 11.5

Открытие файла для чтения или записи без соответствующих привилегий доступа к файлу (это зависит от операционной системы).

Режимы	Описание
r	Открыть файл для чтения.
w	Создать файл для записи. Если файл уже существует, его содержимое пропадает.
a	Добавление: открывает или создает файл для записи в конец файла.
r+	Открывает файл для обновления (чтение и запись).
w+	Создает файл для обновления. Если файл уже существует, его содержимое пропадает.
a+	Добавление: открывает или создает файл для обновления: запись производится в конец файла.

Рис. 11.6. Режимы открытия файлов

Распространенная ошибка программирования 11.6

Открытие файла для записи при отсутствии свободного места на диске. На рис. 11.6 перечислены режимы, в которых могут быть открыты файлы.

Распространенная ошибка программирования 11.7

Неверный выбор режима открытия файла может привести к разрушительным ошибкам. Например, если открыть файл в режиме записи ("w") вместо режима обновления ("r+"), то это приведет к полной потере содержимого файла.

Хороший стиль программирования 11.3

Открывайте файлы только для чтения (без возможности обновления), если записанные в файле данные не должны изменяться. Это позволит избежать неумышленного изменения содержимого файла. Это еще один пример применения правила минимальных привилегий.

11.5. Чтение данных из файла последовательного доступа

Данные хранятся в файлах для того, чтобы их можно было в случае необходимости извлечь и обработать. В предыдущем разделе было показано, как создать файл с последовательным доступом. В этом разделе мы обсудим, как прочитать данные из такого файла.

Программа на рис. 11.7 считывает записи из файла "**clients.dat**", созданного программой на рис. 11.3, и выводит содержимое записей на печать. Оператор

```
FILE *cfPtr;
```

объявляет, что **cfPtr** является указателем на структуру **FILE**. Строка

```
if ((cfPtr = fopen("clients.dat", "r")) == NULL)
```

открывает файл "**clients.dat**" для чтения ("**r**") и проверяет успешность выполнения данной операции (т.е. что **fopen** не возвратила **NULL**). Оператор

```
fscanf(cfPtr, "%d%s%f", &account, name, &balance);
```

считывает «запись» из файла. Функция **fscanf** — это аналог функции **scanf**, за исключением того, что **fscanf** в качестве аргумента передается еще указатель файла, из которого считываются данные. После того, как рассмотренный выше оператор выполнится в первый раз, **account** получит значение **100**, **name** получит значение "**Jones**" и **balance** значение **24.98**. Каждый раз, когда исполняется второй оператор **fscanf**, из файла считывается следующая запись и **account**, **name** и **balance** принимают новые значения. При достижении конца файла он закрывается и выполнение программы завершается.

Для того чтобы считать данные из файла последовательного доступа, программа обычно начинает чтение с начала файла и считывает все данные последовательно до тех пор, пока они не закончатся. В ходе выполнения программы может возникнуть необходимость обработать данные из файла последовательного доступа несколько раз (начиная каждый раз с первой записи). Выполнение оператора

```
rewind(cfPtr);
```

приводит к тому, что *указатель позиции файла* — показывающий номер байта, который должен быть считан следующим — будет перемещен в начало (т.е. нулевой байт) файла, на который указывает **cfPtr**. Указатель позиции файла — не указатель в обычном понимании. Это целое значение, определяющее байт в файле, который будет использован при следующей процедуре чтения или записи. Его еще иногда называют *смещением*. Указатель позиции является элементом связанной с файлом структуры **FILE**.

```
/* Чтение и распечатка последовательного файла */
#include <stdio.h>

main()
{
   int account;
   char name[30];
   float balance;
   FILE *cfPtr;  /* cfPtr = указатель файла clients.dat */

   if ((cfPtr = fopen("clients.dat", "r")) == NULL)
      printf("File could not be opened\n");
   else {
       printf("%-10s%-13s%s\n", "Account", "Name", "Balance");
       fscanf(cfPtr, "%d%s%f", &account, name, &balance);

       while (!feof(cfPtr)) {
          printf("%-10d%-13s%7.2f\n", account, name, balance);
          fscanf(cfPtr, "%d%s%f", &account, name, &balance);
       }

       fclose(cfPtr);
   }

   return 0;
}
```

```
Account    Name         Balance
100        Jones          24.98
200        Doe           345.67
300        White           0.00
400        Stone         -42.16
500        Rich          224.62
```

Рис. 11.7. Чтение и вывод на печать данных из файла последовательного доступа

Теперь можно представить программу (рис. 11.8), которая позволяет менеджеру по кредиту получить список клиентов с нулевым сальдо (клиентов, которые не должны денег), клиентов с положительным сальдо (которым компания должна какую-то сумму денег) и клиентов с отрицательным сальдо (которые должны компании деньги за уже полученные товары и услуги).

Отметим, что данные в этом файле с последовательным доступом не могут быть изменены без риска разрушить другие хранящиеся в файле данные. Например, если имя "**White**" необходимо поменять на "**Worthington**", то нельзя просто взять и переписать старое имя. Запись для **White** была введена в файл как

```
300 White 0.00
```

Если перезапись с новым именем будет сделана в том же месте файла, то она будет выглядеть так:

```
300 Worthington 0.00
```

Новая запись больше, чем первоначальная. Символы после второго "**o**" в "**Worthington**" будут записаны уже поверх следующей записи в файле. Проблема в данном случае состоит в том, что при форматированном вводе/выводе с использованием **fprintf** и **fscanf** поля — а следовательно, и записи — могут иметь переменную длину. Например, 7, 14, –117, 2074 и 27383 являются це-

лыми типа **int** и занимают в памяти компьютера одинаковое пространство, однако при выводе их на экран или при записи на диск с помощью **fprintf** они будут представлены полями разной длины.

```
/*Программа для справок по кредиту */
#include <stdio.h>

main()
{
   int request, account;
   float balance;
   char name[30];
   FILE *cfPtr;

   if ((cfPtr = fopen("clients.dat", "r")) == NULL)
      printf("File could not be opened\n");
   else {
      printf("Enter request\n"
             " 1 - List accounts with zero balances\n"
             " 2 - List accounts with credit balances\n"
             " 3 - List accounts with debit balances\n"
             " 4 - End of run\n? ");
      scanf("%d", &request);

      while (request != 4) {
         fscanf(cfPtr, "%d%s%f", &account, name, &balance);

         switch (request) {
            case 1:
               printf("\nAccounts with zero balances:\n");

               while(!feof(cfPtr)) {

                  if (balance ==0)
                     printf("%-10d%-13s%7.2f\n",
                            account, name, balance);

                  fscanf(cfPtr, "%d%s%f",
                         &account, name, &balance);
               }

               break;
            case 2:
               printf("\nAccounts with credit balances:\n");

               while (!feof(cfPtr)) {

                  if (balance < 0)
                     printf("%-10d%-13s%7.2f\n",
                            account, name, balance);
                  fscanf(cfPtr, "%d%s%f",
                         &account, name, &balance);
               }
               break;
```

Рис. 11.8. Программа для вывода справки о кредите (часть 1 из 2)

```
            case 3:
               printf("\nAccounts with debit balances:\n");

               while (!feof(cfPtr)) {

                  if (balance >0)
                     printf("%-10d%-13s%7.2f\n",
                            account, name, balance);

                  fscanf(cfPtr, "%d%s%f",
                         &account, name, &balance);
               }
               break;
         }

         rewind(cfPtr);
         printf("\n? ");
         scanf("%d", &request);
      }

      printf("End of run.\n");
      fclose(cfPtr);
   }

   return 0;
}
```

Рис. 11.8. Программа для вывода справки о кредите (часть 2 из 2)

```
Enter request
 1 - List accounts with zero balances
 2 - List accounts with credit balances
 3 - List accounts with debit balances
 4 - End of run
? 1

Accounts with zero balances:
300        White           0.00

? 2

Accounts with credit balances:
400        Stone         -42.16

? 3

Accounts with debit balances:
100        Jones          24.98
200        Doe           345.67
500        Rich          224.62

? 4
End of run
```

Рис. 11.9. Пример вывода справки о кредите, выполненного программой на рис. 11.8

Поэтому последовательный доступ с помощью **fprintf** и **fscanf** обычно не используется для обновления записей в файле. Вместо этого переписывают весь файл целиком. Чтобы выполнить упоминавшееся выше изменение имени, необходимо скопировать записи, предшествующие **300 White 0.00**, в новый файл, после чего ввести исправленную запись, а затем дописать в новый файл все записи после **300 White 0.00**. Таким образом, ради того, чтобы изменить одну запись, придется обработать все записи файла.

11.6. Файлы произвольного доступа

Как уже отмечалось ранее, созданные с помощью функции форматированного вывода **fprintf** записи не обязаны иметь одинаковую длину. Напротив, отдельные записи в *файле с произвольным доступом* обычно имеют фиксированную длину, что позволяет получить к ним непосредственный (и следовательно, быстрый) доступ без поиска по всем записям. Поэтому файлы с произвольным доступом используются в системах бронирования мест на самолеты, банковских системах, системах складского учета и других *системах обработки транзакций*, требующих быстрого доступа к данным. Существуют другие способы обеспечения произвольного доступа к файлам, но мы ограничим наше обсуждение подходом, использующим записи с фиксированной длиной, ввиду его простоты и наглядности.

По причине того, что все записи файла с произвольным доступом имеют равную длину, можно вычислить точное положение записи относительно начала файла как функцию ключа записи. Вскоре мы увидим, каким образом это помогает получить мгновенный доступ к конкретной записи, даже в случае, когда размер файла очень велик.

Рис. 11.10 иллюстрирует один из способов организации файла произвольного доступа. Такой файл подобен составу из грузовых вагонов, некоторые вагоны пустые, а некоторые с каким-то грузом. Но каждый вагон состава имеет одну и ту же длину.

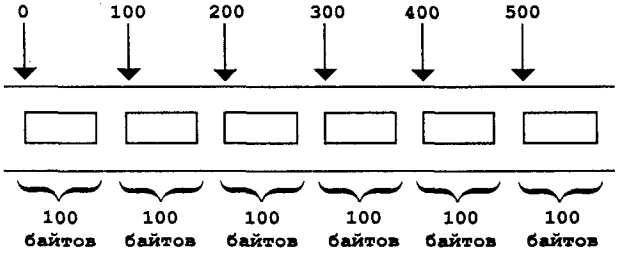


Рис. 11.10. Вид файла произвольного доступа с записями постоянной длины

В файл с произвольным доступом можно вставлять новые данные, не разрушая при этом те, что уже находятся в файле. Ранее сохраненные данные можно также обновить или удалить без переписывания всего файла. В следующих разделах мы объясним, как создавать файл произвольного доступа, вводить данные, считывать данные как последовательно, так и в произвольном порядке, обновлять данные и удалять данные, которые больше не нужны.

11.7. Создание файла произвольного доступа

Функция **fwrite** пересылает в файл заданное число байт начиная с указанного адреса памяти. Данные записываются с того места в файле, которое обозначено указателем позиции файла. Функция **fread** пересылает заданное число байт из места в файле, определенного указателем позиции файла, в массив в памяти, начинающийся с указанного адреса. Теперь при записи целого числа вместо оператора

```
fprintf(fPtr, "%d", number);
```

который может напечатать от одной до одиннадцати цифр (десять цифр плюс знак, требующие по одному байту памяти каждый) для четырехбайтного целого, мы можем применить оператор

```
fwrite(&number, sizeof(int), 1, fPtr);
```

который всегда записывает четыре байта (или два байта для систем с двухбайтовым целым) из переменной **number** в файл, определяемый **fPtr** (далее мы кратко объясним смысл аргумента **1**). В дальнейшем, вызвав **fread**, можно прочитать эти четыре байта в переменную **number**. Хотя **fread** и **fwrite** считывают и записывают данные, такие как целые числа, в формате с фиксированным, а не переменным, размером, данные, с которыми они оперируют, обрабатываются в форме «сырых данных» (то есть в виде байтов), а не в привычном для человека формате, с которым оперируют **printf** и **scanf**.

Функции **fwrite** и **fread** дают возможность чтения и записи массивов данных с диска и на диск. Третий аргумент как в **fread**, так и в **fwrite** — это число элементов массива, которые необходимо считать с диска или записать на диск. Представленный выше вызов функции **fwrite** записывает на диск одно целое, поэтому третий аргумент равен **1** (как если бы записывался один элемент массива).

Программы обработки файлов редко записывают в файл единственное поле. Обычно они записывают за один раз переменную типа **struct**, как будет показано в приведенных ниже примерах.

Рассмотрим следующую формулировку задачи:

Создать программу для работы со счетами клиентов компании, способную хранить до ста записей с фиксированной длиной. Каждая запись должна состоять из номера счета, который будет использоваться как ключ, имени, фамилии и сальдо. В программе должны быть предусмотрены возможности: обновлять счета, вставлять записи с новыми счетами, удалять счет и выводить список всех записей со счетами в отформатированном виде, пригодном для печати, в текстовый файл. Использовать для работы файл с произвольным доступом.

В следующих нескольких разделах описаны технические приемы, необходимые для создания программы работы со счетами. В программе на рис. 11.11 показано, как открыть файл произвольного доступа, определить формат записи с помощью **struct**, записать данные на диск и закрыть файл. Эта программа, используя функцию **fwrite**, инициализирует все сто записей в файле "**credit.dat**" пустой структурой **struct**. Каждая пустая **struct** содержит **0** для номера счета, **NULL** (заданный пустыми кавычками) для фамилии, **NULL** для имени и **0.0** для баланса. Файл, инициализированный подобным образом, резервирует на диске место под данные о клиентах и дает возможность определить, содержит ли конкретная запись данные.

```
/* Последовательное создание файла с произвольным доступом */
#include <stdio.h>

struct clientData {
   int acctNum;
   char lastName[15];
   char firstName[10];
   float balance;
};

main()
{
   int i;
   struct clientData blankClient = {0, "", "", 0.0};
   FILE *cfPtr;

   if ((cfPtr = fopen("credit.dat", "w")) == NULL)
      printf("File could not be opened.\n");
   else {

      for (i = 1; i <= 100; I++)
         fwrite(&blankClient,
                sizeof(struct clientData), 1, cfPtr);

      fclose(cfPtr);
   }

   return 0;
}
```

Рис. 11.11. Создание файла с произвольным доступом

Функция **fwrite** записывает блок (заданное число байтов) данных в файл. В нашей программе в результате выполнения оператора

```
fwrite(&blankClient, sizeof(struct clientData), 1, cfPtr);
```

структура **blankClient** размером **sizeof(struct clientData)** будет записана в файл, указанный **cfPtr**. Операция **sizeof** возвращает размер в байтах для объекта, содержащегося в скобках (в данном случае — это **struct clientData**). Операция **sizeof** *является исполняемой во время компиляции одноместной операцией*, возвращающей целое число без знака. Операция **sizeof** может использоваться для определения размера в байтах для любого типа данных или выражения. К примеру, **sizeof(int)** используется для того, чтобы определить, хранится целое в двух или в четырех байтах для данного конкретного компьютера.

Совет по повышению эффективности 11.2

Многие программисты ошибочно думают, что **sizeof** является функцией, и что использование ее приводит к увеличению времени исполнения программы, связанному с вызовом функции. Однако никакого увеличения времени не происходит, поскольку **sizeof** является операцией, выполняемой во время компиляции.

Функция **fwrite** может в действительности использоваться для записи нескольких элементов массива объектов. Чтобы записать несколько элементов массива, программист использует указатель на массив, как первый аргумент в

вызове **fwrite**, а количество элементов, которое необходимо записать, задает третьим аргументом. В предыдущем операторе **fwrite** использовалась для записи одного объекта, который не являлся элементом массива. Запись одного объекта эквивалентна записи одного элемента массива, поэтому в вызове **fwrite** стоит **1**.

11.8. Произвольная запись данных в файл произвольного доступа

Программа на рис 11.12 записывает данные в файл "**credit.dat**". Для хранения данных в специально определенных местах файла используется комбинация **fseek** и **fwrite**. Функция **fseek** устанавливает указатель позиции файла в заданное положение, после чего **fwrite** записывает данные. Образец выполнения программы показан на рис. 11.13.

```
/* Запись в файл с произвольным доступом */
#include <stdio.h>

struct clientData {
   int acctNum;
   char lastName[15];
   char firstName[10];
   float balance;
};

main()
{
   FILE *cfPtr;
   struct clientData client;

   if ((cfPtr = fopen("credit.dat", "r+")) == NULL)
      printf("File could not be opened.\n");
   else {
      printf("Enter account number"
             " (1 to 100, 0 to end input)\n? ");
      scanf("%d", &client.acctNum);

      while (client.acctNum !=0) {
         printf("Enter lastname, firstname, balance\n? ");
         scanf("%s%s%f", &client.lastName,
               &client.firstName, &client. balance);
         fseek(cfPtr, (client.acctNum -1) *
                sizeof(struct clientData), SEEK_SET);
         fwrite(&client, sizeof(struct clientData), 1, cfPtr);
         printf("Enter account number\n? ");
         scanf("%d", &client.acctNum);
      }
   }

   fclose(cfPtr);

   return 0;
}
```

Рис. 11.12. Запись данных в файл произвольного доступа

```
Enter account number (1 to 100, 0 to end input)
? 37
Enter lastname, firstname, balance
? Barker Doug 0.00
Enter account number
? 29
Enter lastname, firstname, balance
? Brown Nancy -24.54
Enter account number
? 96
Enter lastname, firstname, balance
? Stone Sam 34.98
Enter account number
? 88
Enter lastname, firstname, balance
? Smith Dave 258.34
Enter account number
? 33
Enter lastname, firstname, balance
? Dunn  Stacey 314.33
Enter account number
? 0
```

Рис. 11.13. Пример выполнения программы на рис. 11.12

Оператор

```
fseek(cfPtr, (accountNum -1) * sizeof(struct clientData),
      SEEK_SET);
```

устанавливает указатель позиции файла, на который ссылается **cfPtr** на байт, определяемый выражением **(accountNum-1) * sizeof(struct clientData)**; значение этого выражения называется *смещением*. Поскольку номер счета — число от 1 до 100, а байты в файле начинаются с нулевого, то при нахождении положения байта из номера счета вычитается 1. Таким образом, для первой записи указатель позиции файла устанавливается на байт 0 файла. Символическая константа **SEEK_SET** показывает, что указатель позиции файла устанавливается относительно начала файла на величину смещения. Рассмотрение приведенного выше оператора показывает, что поиск в файле счета с номером 1 устанавливает указатель позиции файла на начало файла, поскольку вычисленное значение равно 0. На рис. 11.14 показан указатель файла, ссылающийся на структуру **FILE** в памяти. Указатель позиции файла показывает, что следующий байт, который будет считан или записан, это пятый байт от начала файла.

Согласно стандарту ANSI прототип функции **fseek** определяется как

```
int fseek(FILE *stream, long int offset, int whence);
```

где **offset** — число байт от положения, задаваемого **whence**, в файле, на который указывает **stream**. Аргумент **whence** может иметь одно из трех значений — **SEEK_SET**, **SEEK_CUR** или **SEEK_END**, — указывающих место в файле, с которого начинается поиск. **SEEK_SET** указывает, что поиск начнется с начала файла; **SEEK_CUR** указывает, что поиск начнется с текущего положения в файле; и наконец, **SEEK_END** указывает, что поиск начнется с конца файла. Эти три символические константы определены в заголовочном файле **stdio.h**.

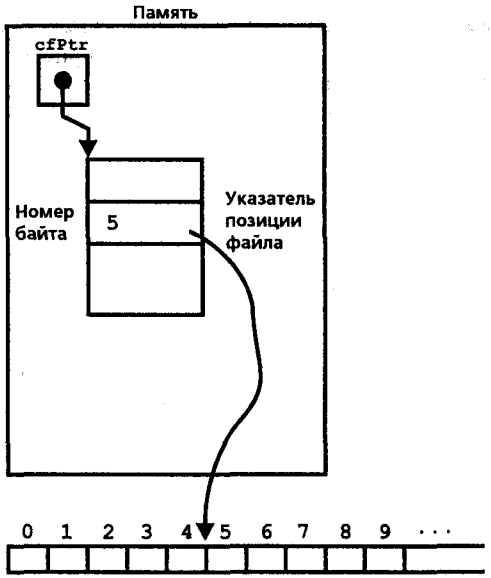


Рис. 11.14. Указатель позиции файла показывает на сдвиг в пять байт от начала файла

11.9. Последовательное чтение данных из файла произвольного доступа

Функция **fread** считывает заданное число байт из файла в память. Например, оператор

```
fread(&client, sizeof(struct clientData), 1, cfPtr);
```

считывает число байтов, равное **sizeof(struct clientData)**, из файла, на который ссылается **cfPtr**, и сохраняет данные в структуре **client**. Байты считываются из файла начиная с места, определенного указателем позиции файла. Функцию **fread** можно использовать и для чтения нескольких элементов массива с фиксированным размером с помощью указателя на массив, в котором будут храниться элементы, и указанием числа элементов, которые необходимо прочитать. Представленный выше оператор определяет, что должен быть прочитан один элемент. Для того чтобы прочитать более одного элемента, необходимо задать соответствующее значение в третьем аргументе функции **fread**.

Программа на рис. 11.15 считывает последовательно каждую запись из файла "**credit.dat**", проверяет все записи на наличие данных и распечатывает содержащие данные записи в форматированном виде. Функция **feof** отслеживает момент, когда достигается конец файла, а функция **fread** передает данные с диска в структуру **client** типа **clientData**.

11.10. Пример: программа обработки транзакций

В этом разделе будет представлена довольно сложная программа обработки транзакций, обрабатывающая файлы произвольного доступа. Она предназначена

для работы с банковскими счетами. Программа позволяет обновлять существующие счета, добавлять новые, уничтожать счета и сохранять список всех текущих счетов в файле текстового формата для вывода на печать. Предполагается, что была выполнена программа на рис. 11.11, создающая файл **credit.dat**.

```
/* Последовательное чтение файла с произвольным доступом */
#include <stdio.h>

struct clientData {
   int acctNum;
   char lastName[15];
   char firstName[10];
   float balance;
};

main()
{
   FILE *cfPtr;
   struct clientData client;

   if ((cfPtr = fopen("credit.dat", "r")) == NULL)
      printf("File could not be opened.\n");
   else {
      printf("%-6s%-16s%-11s%10s\n", "Acct", "Last Name",
             "First Name", "Balance");

      while (!feof(cfPtr)) {
         fread(&client, sizeof(struct clientData), 1, cfPtr);

         if (client.acctNum !=0)
            printf("%-6d%-16s%-11s%10.2f\n",
                   client.acctNum, client.lastName,
                   client.firstName, client.balance);
      }
   }

   fclose(cfPtr);

   return 0;
}
```

```
Acct  Last Name      First Name  Balance
29    Brown          Nancy        -24.54
33    Dunn           Stacey       314.33
37    Barkey         Doug           0.00
88    Smith          Dave         258.34
96    Stone          Sam           34.98
```

Рис. 11.15. Последовательное считывание из файла произвольного доступа

Программа имеет пять опций. Опция 1 вызывает функцию **textFile** для сохранения отформатированного списка всех счетов в текстовом файле, с именем **accounts.txt**, который позднее можно вывести на печать. Функция использует **fread** и метод последовательного доступа к файлу, который был использован ранее в программе, представленной на рис. 11.15. После выбора опции 1 создается файл **accounts.txt**, содержащий следующий текст:

```
Acct        Last Name       First Name      Balance
29          Brown           Nancy            -24.54
33          Dunn            Stacey           314.33
37          Barkey          Doug               0.00
88          Smith           Dave             258.34
96          Stone           Sam               34.98
```

Опция 2 вызывает функцию **updateRecord** для обновления счета. Функция предназначена для обновления уже существующих записей, поэтому сначала она проверяет, не является ли указанная пользователем запись пустой. Запись считывается в структуру **client** с помощью функции **fread**, после чего элемент **acctNum** сравнивается с 0. Если он равен 0, то запись не содержит никакой информации, и выдается сообщение, что запись пустая. После чего на экране вновь появляется меню выбора. Если запись содержит информацию, функция **updateRecord** вводит сумму транзакции, вычисляет новый баланс и переписывает соответствующую запись в файле. Типичный результат исполнения опции 2 выглядит следующим образом:

```
Enter account to update (1 - 100): 37
37    Barker          Doug              0.00

Enter charge (+) or payment (-): +87.99
37    Barker          Doug             87.99
```

Опция 3 вызывает функцию **newRecord** для добавления в файл нового счета. Если пользователь вводит уже существующий номер счета, **newRecord** выдает сообщение об ошибке, что данная запись уже содержит информацию, после чего на экран вновь выводится меню выбора. Эта функция использует тот же процесс добавления нового счета, что и программа на рис. 11.12. Типичный результат исполнения опции 3 выглядит следующим образом:

```
Enter new account number (1 - 100): 22
Enter lastname, firstname, balance
? Johnson Sarah 247.45
```

Опция 4 вызывает функцию **deleteRecord** для удаления записи из файла. Пользователя просят ввести номер счета, который необходимо удалить, после чего выполняется повторная инициализация записи. Если счет не содержит информации, **deleteRecord** выдает сообщение об ошибке, что указанного счета не существует. Опция 5 прерывает выполнение программы. Полный текст программы представлен на рис. 11.16. Заметим, что файл "**credit.dat**" открыт для обновления (чтения и записи) со спецификацией режима "**r+**".

```
/* Программа последовательно читает файл произвольного доступа, *
 * обновляет данные, ранее записанные в файл, создает новые     *
 * данные для помещения в файл, и удаляет данные,               *
 * уже имеющиеся в файле.                                       */

#include <stdio.h>

struct clientData {
   int acctNum;
   char lastName[15];
   char firstName[10];
   float balance;
};
```

Рис. 11.16. Программа работы с банковскими счетами (часть 1 из 4)

```
int enterChoice(void);
void textFile(FILE *);
void updateRecord(FILE *);
void newRecord(FILE *);
void deleteRecord(FILE *);

main()
{
   FILE *cfPtr;
   int choice;

   if ((cfPtr = fopen("credit.dat", "r+")) == NULL)
      printf("File could not be opened.\n");
   else {

      while ((choice = enterChoice()) != 5) {

         switch (choice) {
            case 1:
               textFile(cfPtr);
               break;
            case 2:
               updateRecord(cfPtr);
               break;
            case 3:
               newRecord(cfPtr);
               break;
            case 4:
               deleteRecord(cfPtr);
               break;
         }

      }
   }

   fclose(cfPtr);
   return 0;
}

void textFile(FILE *readPtr)
{
   FILE *writePtr;
   struct clientData client;

   if ((writePtr = fopen("accounts.txt", "w")) == NULL)
      printf("File could not be opened.\n");
   else {
      rewind(readPtr);
      fprintf(writePtr, "%-6s%-16s%-11s%10s\n",
              "Acct", "Last Name", "First Name", "Balance");

      while (!feof(readPtr)) {
         fread(&client, sizeof(struct clientData), 1, readPtr);
```

Рис. 11.16. Программа работы с банковскими счетами (часть 2 из 4)

```
         if (client.acctNum !=0)
            fprintf(writePtr, "%-6d%-16s%-11s%10.2f\n",
                    client.acctNum, client.lastName,
                    client.firstName, client.balance);
      }
   }

   fclose(writePtr);
}

void updateRecord(FILE *fPtr)
{
   int account;
   float transaction;
   struct clientData client;
   printf("Enter account to update (1 - 100): ");
   scanf("%d", &account);
   fseek(fPtr, (account - 1) * sizeof(struct clientData),
         SEEK_SET);
   fread(&client, sizeof(struct clientData), 1, fPtr);

   if (client.acctNum == 0)
      printf("Acount #%d has no information.\n",account);
   else {
      printf("%-6d%-16s%-11s%10.2f\n\n",
             client.acctNum, client.lastName,
             client.firstName, client.balance);
      printf("Enter chrge (+) or payment(-): ");
      scanf("%f", &transaction);
      client.balance += transaction;
      printf("%-6d%-16s%-11s%10.2f\n",
             client.acctNum, client.lastName,
             client.firstName, client.balance);
      fseek(fPtr, (account-1) * sizeof(struct clientData),
            SEEK_SET);
      fwrite(&client, sizeof(struct clientData), 1, fPtr);
   }
}

void deleteRecord(FILE *fPtr)
{
struct clientData client, blankClient = {0, "", "", 0};
   int accountNum;

   printf("Enter account number to delete (1-100): ");
   scanf("%d", &accountNum);
   fseek(fPtr, (accountNum - 1) * sizeof(struct clientData),
         SEEK_SET);
   fread(&client, sizeof(struct clientData), 1, fPtr);

   if (client.acctNum == 0)
      printf("Accout %d does not exist.\n", accountNum);
   else {
      fseek(fPtr, (accountNum - 1) * sizeof(struct clientData),
            SEEK_SET);
```

Рис. 11.16. Программа работы с банковскими счетами (часть 3 из 4)

```
      fwrite(&blankClient, sizeof(struct clientData), 1, fPtr);
   }
}

void newRecord(FILE *fPtr)
{
   struct clientData client;
   int accountNum;
   printf("Enter new account number (1-100): ");
   scanf("%d", &accountNum);
   fseek(fPtr, (accountNum -1) * sizeof(struct clientData),
         SEEK_SET);
   fread(&client, sizeof(struct clientData), 1, fPtr);

   if (client.acctNum !=0)
      printf("Account #%d already contains information.\n",
             client.acctNum);
   else {
      printf("Enter lastname, firstname, balance\n? ");
      scanf("%s%s%f", &client.lastName, &client.firstName,
            &client.balance);
      client.acctNum = accountNum;
      fseek(fPtr, (client.acctNum -1) *
            sizeof(struct clientData), SEEK_SET);
      fwrite(&client, sizeof(struct clientData), 1, fPtr);
   }
}

int enterChoice(void)
{
   int menuChoice;

   printf("\nEnter your choice\n"
          "1 - store a formatted text file of accounts called\n"
          "    \"accounts.txt\" for printing\n"
          "2 - update an account\n"
          "3 - add a new account\n"
          "4 - delete an account\n"
          "5 - end program\n? ");
   scanf("%d", &menuChoice);
   return menuChoice;
}
```

Рис. 11.16. Программа работы с банковскими счетами (часть 4 из 4)

Резюме

- Все элементы данных, обрабатываемые компьютером, сводятся к комбинации нолей и единиц.
- Значение наименьшего элемента данных, к которому может обращаться компьютер, равно **0** или **1**. Этот элемент данных называется *бит* (по-английски *bit* — сокращение от «*bi*nary digi*t*» — число, способное принимать одно из двух значений).

- Цифры, буквы и специальные знаки называют символами. Множество всех символов, которые можно использовать для написания программ и представления элементов данных на конкретном компьютере, называется его набором символов. Каждый символ представляется в компьютере комбинацией из восьми нулей и единиц (называемой байтом).
- Полем называется группа символов, передающая значение.
- Записью называется группа связанных между собой полей.
- По крайней мере одно поле в каждой записи обычно выбирается в качестве ключа записи. Ключ записи идентифицирует запись, как принадлежащую конкретному лицу или объекту.
- Наиболее популярный тип организации записей в файле называется последовательным файлом; в нем осуществляется поочередный доступ к записям, пока не будут найдены требуемые данные.
- Группу связанных между собой файлов иногда называют базой данных. Набор программ, разработанных для создания и управления базами данных, называется системой управления базами данных (DBMS).
- Язык C рассматривает любой файл как последовательный поток байтов.
- В начале исполнения программы C автоматически открывает три файла и связанные с ними потоки: стандартный ввод, стандартный вывод и стандартный файл ошибок.
- Стандартному вводу, стандартному выводу и стандартному файлу ошибок присваиваются соответственно указатели файлов **stdin**, **stdout** и **stderr**.
- Функция **fgetc** считывает символ из указанного файла.
- Функция **fputc** записывает символ в указанный файл.
- Функция **fgets** считывает строку из указанного файла.
- Функция **fputs** записывает строку в указанный файл.
- **FILE** является структурой, тип которой определен в заголовочном файле **stdio.h**. Программисту, чтобы работать с файлами, нет необходимости знать, как устроена эта структура. При открытии файла структура **FILE** возвращает указатель на файл.
- Функция **fopen** принимает два аргумента: имя файла и режим, в котором должен быть открыт файл, и открывает файл. Если файл, открываемый для записи, уже существует, его содержимое уничтожается без предупреждения. Если файла не существовало и файл открывается для записи, **fopen** создает файл.
- Функция **feof** определяет, был ли установлен индикатор конца файла.
- Функция **fprintf** — аналог **printf** за исключением того, что **fprintf** получает в качестве аргумента еще и указатель на файл, в который будут записаны данные.
- Функция **fclose** закрывает файл, на который указывает передаваемый ей аргумент.

- Чтобы создать файл или уничтожить содержимое файла, в который данные были записаны ранее, откройте файл для записи ("**w**"). Чтобы прочитать существующий файл, откройте файл для чтения ("**r**"). Чтобы добавить записи в конец существующего файла, откройте файл для добавления ("**a**"). Чтобы открыть файл так, чтобы с ним можно было выполнять операции записи и чтения, откройте файл для обновления в одном из режимов "**r+**", "**w+**" или "**a+**". Режим "**r+**" просто открывает файл для чтения и записи. Режим "**w+**" создает файл, если он не существовал ранее, и стирает текущее содержание файла в противном случае. Режим "**a+**" создает файл, если он не существует, и производит запись в конец файла.
- Функция **fscanf** — аналог **scanf**, за исключением того, что **fscanf** получает в качестве дополнительного аргумента указатель на файл (обычно отличный от **stdin**), из которого будут считываться данные.
- Выполнение функции **rewind** приводит к перемещению указателя позиции в начало указанного файла.
- Процесс произвольного доступа к файлу используется для прямого доступа к записям.
- Для упрощения произвольного доступа данные хранятся в записях фиксированной длины. Так как все записи имеют одинаковую длину, компьютер может быстро вычислять (как функцию ключа записи) точное положение записи относительно начала файла.
- В файл произвольного доступа новые данные можно легко добавить без разрушения уже хранящихся там данных. Данные, хранящиеся в файле с фиксированной длиной записи, можно также изменять или удалять без переписывания всего файла.
- Функция **fwrite** записывает блок (заданное число байт) данных в файл.
- Исполняемая во время компиляции операция **sizeof** возвращает размер операнда в байтах.
- Функция **fseek** устанавливает указатель позиции файла в заданную позицию, исходя из позиции, от которой начинается поиск. Поиск начинается с одной из трех позиций — **SEEK_SET** начинает поиск с начала файла, **SEEK_CUR** с текущей позиции, а **SEEK_END** с конца файла.
- Функция **fread** считывает блок (заданное число байт) данных из файла.

Терминология

fclose
feof
fgetc
fgets
fopen
fprintf
fputc
fputs
fread
fscanf
fseek
fwrite
rewind
SEEK_CUR
SEEK_END
SEEK_SET
stderr (стандартная ошибка)
stdin (стандартный ввод)
stdout (стандартный вывод)
алфавитный порядок
база данных
байт

бит
буква
буквенное поле
буквенно-цифровое поле
двоичная цифра
десятичная цифра
закрытие файла
запись
иерархия данных
имя файла
индикатор конца файла
ключ записи
конец файла
набор символов
нули и единицы
открытие файла
параметр числа записей
поле
поток
произвольный доступ
режим открытия файла
режим открытия файла **a**
режим открытия файла **a+**
режим открытия файла **r**
режим открытия файла **r+**
режим открытия файла **w**
режим открытия файла **w+**
символ
символьное поле
система управления базами данных
смещение
структура **FILE**
указатель позиции файла
указатель файла
файл
файл последовательного доступа
файл произвольного доступа
форматированный ввод/вывод
целое число
численное поле

Распространенные ошибки программирования

11.1. Открытие ранее созданного файла для записи ("**w**"), тогда как на самом деде пользователь хочет сохранить дынные, находящиеся в файле; содержимое файла уничтожается без предупреждения.

11.2. Забывают открыть файл, перед тем как ссылаться на него в программе.

11.3. Ссылаясь на файл, используют неверный указатель файла.

11.4. Открытие для чтения несуществующего файла.

11.5. Открытие файла для чтения или записи без соответствующих привилегий доступа к файлу (это зависит от операционной системы).

11.6. Открытие файла для записи при отсутствии свободного места на диске. На рис. 11.6 перечислены режимы, в которых могут быть открыты файлы.

11.7. Неверный выбор режима открытия файла может привести к разрушительным ошибкам. Например, если открыть файл в режиме записи ("**w**") вместо режима обновления ("**r+**"), то это приведет к полной потере содержимого файла.

Хороший стиль программирования

11.1. Убедитесь, что вызовы функций, работающих с файлами, содержат корректные указатели файлов.

11.2. Явно закрывать каждый файл, как только выяснится, что программа больше не будет обращаться к файлу.

11.3. Открывайте файлы только для чтения (без возможности обновления), если записанные в файле данные не должны изменяться. Это позволит избежать неумышленного изменения содержимого файла. Это еще один пример применения правила минимума привилегий.

Советы по повышению эффективности

11.1. Закрывая файл, мы освобождаем ресурсы, которые, возможно, требуются другим пользователям или программам.

11.2. Многие программисты ошибочно думают, что **sizeof** является функцией, и что использование ее приводит к увеличению времени исполнения программы, связанному с вызовом функции. Однако никакого увеличения времени не происходит, поскольку **sizeof** является операцией, выполняемой во время компиляции.

Советы по переносимости программ

11.1. Структура **FILE** зависит от типа операционной системы (т. е. элементы структуры меняются в зависимости от того, как каждая система обращается со своими файлами).

Упражнения для самоконтроля

11.1. Заполните пробелы в каждом из следующих предложений.

a) В конечном счете все элементы данных, обрабатываемых компьютером, сводятся к комбинации _____ и _____.

b) Наименьший элемент данных, который может обработать компьютер, называется _____.

c) _____ — это группа связанных между собой записей.

d) Цифры, буквы и специальные знаки в совокупности называются _____.

e) Группа связанных между собой файлов называется _____.

f) Функция _____ закрывает файл.

g) Оператор _____ считывает данные из файла, так же как **scanf** считывает из **stdin**.

h) Функция _____ считывает символ из заданного файла.

i) Функция _____ считывает строку из заданного файла.

j) Функция _____ открывает файл.

k) Функция _____ обычно применяется для чтения данных из файлов с произвольным доступом

l) Функция _____ переустанавливает указатель позиции файла в заданное положение.

11.2. Установите, верны или неверны следующие утверждения (если неверны, объясните, почему):

a) Функцию **fscanf** нельзя использовать для чтения данных со стандартного ввода.

b) Программист должен явно использовать **fopen** для открытия потоков стандартного ввода, стандартного вывода и стандартного файла ошибок.

c) Программа должна явно вызывать функцию **fclose** для того, чтобы закрыть файл.

d) Если указатель позиции файла установлен в файле с последовательным доступом на положение, отличное от его начала, то, чтобы начать чтение с начала файла, его необходимо закрыть и вновь открыть.

e) Функция **fprintf** может писать в стандартный вывод.

f) Данные в файле последовательного доступа всегда обновляются без переписывания остальных данных.

g) В файле произвольного доступа нет необходимости просматривать все записи для того, чтобы найти определенную запись.

h) Записи в файлах произвольного доступа имеют разную длину.

i) Функция **fseek** может производить поиск только от начала файла.

11.3. Напишите одиночные операторы C, выполняющие каждое из следующих действий. Предполагается, что все эти операторы используются в одной и той же программе.

a) Напишите оператор, который открывает файл "**oldmast.dat**" на чтение и присваивает возвращаемый указатель файла **ofPtr**.

b) Напишите оператор, который открывает файл "**trans.dat**" на чтение и присваивает возвращаемый указатель файла **tfPtr**.

c) Напишите оператор, который открывает файл "**newmast.dat**" на запись (и создание) и присваивает возвращаемый указатель файла **nfPtr**.

d) Напишите оператор, считывающий запись из "**oldmast.dat**". Запись состоит из целого числа **accountNum**, строки **name** и числа с плавающей точкой **currentBalance**.

e) Напишите оператор, считывающий запись из "**trans.dat**". Запись состоит из целого числа **accountNum** и числа с плавающей запятой **dollarAmount**.

f) Напишите оператор, производящий запись в "**newmast.dat**". Запись состоит из целого числа **accountNum**, строки **name** и числа с плавающей запятой **currentBalance**.

11.4. Найдите ошибку в каждом из следующих фрагментов программы. Объясните, как можно эту ошибку исправить.

a) Файл, на который ссылается **fPtr("payables.dat")**, не был открыт.

```
fprintf(fPtr, "%d%s%d\n", account, company, amount);
```

b)
```
open("receive.dat", "r+");
```

c) Следующий оператор должен считывать запись из файла "**payables.dat**". Указатель файла **payPtr** ссылается на этот файл, а указатель файла **recPtr** ссылается на файл "**receive.dat**".

```
fscanf(recPtr, "%d%s%d\n", &account, company, &amount);
```

d) Файл "**tools.dat**" должен быть открыт для добавления данных в файл без уничтожения уже существующих данных.

```
if ((tfPtr = fopen("tools.dat", "w")) != NULL)
```

e) Файл "**courses.dat**" необходимо открыть для добавления без изменения текущего содержимого файла.

```
if ((cfPtr = fopen("courses.dat", "w+")) != NULL)
```

Ответы на упражнения для самоконтроля

11.1. a) нулей, единиц b) бит, c) файл, d) символами, e) базой данных, f) **fclose**, g) **fscanf**, h) **gets** или **fgetc**, i) **fgets**, j) **fopen**, k) **fread**, l) **fseek**.

11.2. a) Неверно. Функцию **fscanf** можно использовать для чтения со стандартного ввода, используя в вызове **fscanf** указатель на поток стандартного ввода **stdin**.

b) Неверно. Эти три потока открываются автоматически, как только начинает исполняться программа.

c) Неверно. Файлы будут закрыты при завершении выполнения программы, однако рекомендуется явно закрывать все файлы с помощью **fclose**.

d) Неверно. Для перемещения указателя позиции файла на начало файла можно использовать функцию **rewind**.

e) Верно.

f) Неверно. В большинстве случаев записи в файлах последовательного доступа имеют разную длину. Следовательно, возможно, что такое обновление записи приведет к тому, что часть других записей будет переписана.

g) Верно.

h) Неверно. Записи в файлах произвольного доступа обычно имеют одинаковую длину.

i) Неверно. Поиск возможен с начала файла, с конца файла и с текущей позиции в файле, определяемой значением указателя позиции файла.

11.3. a)
```
ofPtr = fopen("oldmast.dat", "r");
```

b)
```
tfPtr = fopen("trans.dat", "r");
```

c)
```
nfPtr = fopen("newmast.dat", "w");
```

d)
```
fscanf(ofPtr, "%d%s%f", &accountNum, name,
       &currentBalance);
```

e)
```
fscanf(tfPtr, "%d%f", &accountNum, &dollarAmount);
```

f)
```
fprintf(nfPtr, "%d%s%.2f", accountNum, name,
        currentBalance);
```

11.4. a) Ошибка: файл "**payables.dat**" не был открыт прежде, чем на него сослались при помощи указателя файла.

Исправление: вызвать **fopen** для открытия "**payables.dat**" в режиме записи, добавления или обновления.

b) Ошибка: вызывается функция **open** вместо **fopen**.

Исправление: использовать функцию **fopen**.

c) Ошибка: оператор **fscanf** использует неверный указатель файла при ссылке на файл "**payables.dat**".

Исправление: использовать указатель **payPtr** при ссылках на "**payables.dat**".

d) Ошибка: содержимое файла будет уничтожено, поскольку файл открыт для записи ("**w**").

Исправление: для того чтобы добавить данные в файл, следует открыть его для обновления ("**r+**") или для добавления ("**a**").

e) Ошибка: файл "**courses.dat**" открыт для обновления в режиме "**w+**", который уничтожает текущее содержимое файла.

Исправление: открыть файл в режиме "**a**".

Упражнения

11.5. Заполните пробелы в каждом из следующих утверждений:

a) Компьютеры сохраняют большие объемы данных на устройствах вторичной памяти, таких как: _____.

b) _____ состоит из нескольких полей.

c) Поле, которое может содержать цифры, буквы и пробелы, называется _____ полем.

d) Для того чтобы облегчить поиск записей в файле, одно поле каждой записи выбирается в качестве _____.

e) Основная масса информации, хранящейся в компьютерных системах, содержится в _____ файлах.

f) Группа связанных символов, передающая значение, называется _____.

g) Указатели файла для трех файлов, которые автоматически открываются C, как только программа начинает исполняться, называются: _____, _____ и _____.

h) Функция _____ записывает символ в указанный файл.

i) Функция _____ записывает строку в указанный файл.

j) Функция _____ в основном используется для записи данных в файл произвольного доступа.

k) Функция _____ перемещает указатель позиции в начало файла.

11.6. Установите, какие из нижеследующих утверждений верны, а какие нет (если утверждение неверно, объясните, почему):

a) Все, даже самые сложные задачи, которые способен выполнять компьютер, по существу представляют собой обработку нулей и единиц.

b) Люди предпочитают иметь дело с битами вместо символов и полей, потому что биты более компактны.

c) Люди записывают программы и данные символами; компьютеры в дальнейшем обрабатывают эти символы как наборы нулей и единиц.

d) Почтовый индекс — пример числового поля.

e) Почтовый адрес в компьютерных приложениях обычно рассматривается как буквенное поле.

f) Элементы обрабатываемых компьютером данных образуют иерархию, в которой элементы данных становятся больше по размеру и сложнее по структуре по мере продвижения от полей к символам, от символов к битам и так далее.

g) Ключ записи определяет запись как принадлежащую особому полю.

h) Большинство организаций хранят свою информацию в виде единого файла для упрощения ее компьютерной обработки.

i) В программах на C ссылки на файл всегда осуществляются по имени.

j) Когда программа создает файл, он автоматически сохраняется компьютером для дальнейших ссылок.

11.7. В упражнении 11.3 мы просили читателя написать ряд отдельных операторов. В действительности эти операторы образует ядро важного типа программ, а именно программ сопоставления файлов. В программах обработки данных для коммерческой сферы обычным является использование нескольких файлов. Например, в программе, предназначенной для работы со счетами клиентов, имеется так называемый главный файл, содержащий подробную информацию о каждом заказчике, такую как имя заказчика, его адрес, номер телефона, задолженность, лимит кредита, условия скидки, условия договора, и, кроме того, краткая сводка последних заказов и поступления платежей.

Если произошла транзакция (то есть совершена покупка и получен денежный перевод), она вносится в файл. В конце каждого делового цикла (то есть месяца для одних компаний, недели для других, а в некоторых случаях дня) файл транзакций (в упражнении 11.3 он назван "**trans.dat**") вводится в главный файл (в упражнении 11.3 "**oldmast.dat**") и, таким образом, производится обновление записей заказов и платежей. После того как все обновления сделаны, главный файл переписывается как новый файл ("**newmast.dat**"), который затем используется в конце следующего делового цикла, чтобы опять произвести процесс обновления.

Программа сопоставления файлов неизбежно сталкивается с некоторыми проблемами, которых не возникает в программах, работающих с одним файлом. Например, сопоставление записей возможно не всегда. В главном файле содержится запись о заказчике, но за текущий деловой период он не делал никаких покупок или платежей и, следовательно, записи в файле транзакций для этого заказчика отсутствуют. А возможна и обратная ситуация, когда заказчик совершил несколько покупок или денежных платежей, но сделал это впервые и вполне возможно, что в компании на данный момент запись в главном файле для этого заказчика отсутствует.

Используя в качестве основы операторы, написанные в упражнении 11.3, напишите полную программу обработки транзакций с сопоставлением файлов. Для целей сопоставления используйте номер счета в каждом файле как ключ. Исходите из того, что каждый

файл является файлом последовательного доступа с записями, хранящимися в порядке возрастания номера счета.

Когда имеется пара записей, которые можно сопоставить (т.е. имеются записи с одним и тем же номером счета в главном файле и файле транзакций), прибавьте сумму в долларах из файла транзакций к текущему балансу главного файла и поместите соответствующую запись в файл **"newmast.dat"**. (Предположим, что в файле транзакций заказам соответствуют положительные денежные суммы, а полученным платежам — отрицательные.) Когда для конкретного счета имеется главная запись, но не существует соответствующей записи в файле транзакций, то главная запись просто переносится в файл **"newmast.dat"**. Когда есть запись в файле транзакций, но нет соответствующей главной записи, программа должна выдавать сообщение **"Unmatched transaction record for account number ..."** (на месте многоточия должен стоять номер счета из записи файла транзакций).

11.8. Напишите простую программу для создания тестовых данных, позволяющих проверить работу программы из упражнения 11.7. Используйте следующий образец данных:

Главный файл:

Номер счета	Имя	Баланс
100	Alan Jones	348.17
300	Mary Smith	27.19
500	Sam Sharp	0.00
700	Suzy Green	-14.22

Файл транзакций:

Номер счета	Сумма
100	27.14
300	62.11
400	100.56
900	82.17

11.9. Запустите программу упражнения 11.7, используя файлы тестовых данных, созданные в упражнении 11.8. Запустите программу из раздела 11.7, чтобы распечатать новый главный файл. Внимательно проверьте результаты.

11.10. Может случиться (и такие случаи не редкость), что некоторые записи транзакций будут иметь одинаковый ключ. Это происходит по той простой причине, что конкретный заказчик может сделать несколько покупок и платежей за деловой период. Перепишите вашу программу с сопоставлением файлов из упражнения 11.7, чтобы предусмотреть возможность обработки нескольких записей транзакций с одинаковым ключом. Измените тестовые данные упражнения 11.8, чтобы включить в них следующие дополнительные записи транзакций:

Номер счета	Сумма
300	83.89
700	80.78
700	1.53

11.11. Напишите операторы, которые выполняют указанные ниже действия. Предполагается, что была определена структура

```
struct person {
   char lastName[15];
   char firstName[15];
   char age[2];
};
```

и что файл уже открыт для записи.

a) Инициализируйте файл "**nameage.dat**" так, чтобы в нем было 100 записей с **lastName** = "**unassigned**", **firstName** = "", и **age** = "**0**".

b) Введите 10 фамилий, имен и возрастов и запишите их в файл.

c) Обновите запись; если в записи нет информации, сообщите пользователю "**No info**".

d) Удалите содержащую информацию запись, заново инициализировав ее.

11.12. Вы владелец магазина инструментов и должны провести инвентаризацию, в результате которой необходимо выяснить, какие инструменты вы имеете, в каком количестве, и стоимость каждого инструмента. Напишите программу, которая инициализирует файл "**hardware.dat**" содержащий 100 пустых записей, дает возможность ввести данные, относящиеся к каждому инструменту, просмотреть список всех имеющихся инструментов, удалить запись, относящуюся к товару, который закончился, и которая позволяет обновить *любую* информацию в файле. Идентификационным номером для инструмента должен служить номер записи. Используйте следующую информацию для того, чтобы начать файл:

Запись №	Название инструмента	Количество	Цена
3	Electric sander	7	57.98
17	Hammer	16	11.99
24	Jig saw	21	11.00
39	Lawn mower	3	79.50
56	Power saw	18	99.99
68	Screwdriver	106	6.99
77	Sledge hammer	11	21.50
83	Wrench	34	7.50

11.13. ***Генератор слова для телефонного номера.*** Стандартный набор кнопок телефона содержит цифры от 0 до 9. Каждая цифра от 2 до 9 имеет связанные с ней три буквы, что отражено в следующей таблице:

Цифра	Буква
2	A B C
3	D E F
4	G H I
5	J K L
6	M N O
7	P R S
8	T U V
9	X Y Z

Многие люди с трудом запоминают номера телефонов, поэтому они используют соответствие между цифрами и буквами, чтобы подобрать слово из семи букв, которое соответствовало бы телефонному номеру. Например, человек, телефонный номер которого 686-2377, может воспользоваться подобной таблицей и подобрать семибуквенное слово «NUMBERS».

Предприниматели часто пытаются получить номер телефона, который было бы легко запомнить их клиентам. Если предприниматель сможет поместить в рекламе простое слово, по которому клиенты могли бы звонить в его контору, тогда, вне всяких сомнений, звонков будет несколько больше.

Каждое слово из семи букв соответствует ровно одному телефонному номеру. Ресторан, желающий увеличить количество заказов на дом, безусловно сможет сделать это, если его номер 825-3688 (т.е. «TAKEOUT»).

Каждому из семизначных номеров соответствует множество слов из семи букв. К сожалению, большинство из них представляет собой бессмысленные комбинации букв. Возможно, однако, что владелец парикмахерской был бы приятно удивлен, узнав, что его телефон 424-7288 соответствует «HAIRCUT». Владелец магазина, торгующего алкоголем, обрадуется, обнаружив, что телефон магазина 233-7226 соответствует «BEERCAN». Ветеринар, телефонный номер которого 738-2273, будет рад узнать, что этот номер соответствует слову «PETCARE».

Напишите программу на C, которая для данного семизначного числа записывает в файл все возможные слова из семи букв, соответствующие этому номеру. Существует 2187 (три в седьмой степени) таких слов. Избегайте телефонных номеров с цифрами 0 и 1.

11.14. Если на вашем компьютере имеется электронный словарь, модифицируйте написанную в упражнении 11.13 программу, чтобы можно было искать слова в словаре. Некоторые комбинации из семи букв

могут состоять из двух или более слов (телефонный номер 843-2677 соответствует «THEBOSS»).

11.15. Измените пример из рис. 8.14, использовав функции **fgetc** и **fputs** вместо **getchar** и **puts**. Программа должна давать пользователю возможность выбирать опции либо чтения со стандартного ввода и записи в стандартный вывод, либо чтения из заданного файла и записи в заданный файл. Если пользователь выбирает второй вариант, то должен ввести имена для входного и выходного файлов.

11.16. Напишите программу, применяющую операцию **sizeof** для определения размеров в байтах различных типов данных на вашей системе. Запишите результаты в файл "**datasize.dat**", чтобы иметь возможность позже их распечатать. Формат файла должен быть следующим:

```
Data type                Size
char                        1
unsigned char               1
short int                   2
unsigned short int          2
int                         4
unsigned int                4
long int                    4
unsigned long int           4
float                       4
double                      8
long double                16
```

Замечание: размер данных для вашего компьютера может отличаться от значений, показанных на рисунке.

11.17. В упражнении 7.19 вы написали программу, моделирующую компьютер, воспринимающий специальный язык, названный Машинным языком Симплетрона (SML). Каждый раз, когда вы хотели запустить SML-программу, ее приходилось вводить в симулятор с клавиатуры. Если вы допускали во время ввода ошибку, симулятор перезапускался и код SML приходилось вводить заново. Было бы замечательно, если бы существовала возможность считывать программы из файла, а не набирать их каждый раз заново. Это позволит сэкономить время и силы, необходимые для запуска программ SML.

a) Модифицируйте симулятор, который вы написали в упражнении 7.19, чтобы можно было считывать программы из файла, имя которого вводится пользователем с клавиатуры.

b) После выполнения программы содержимое регистров и памяти симулятора выводятся на экран. Было бы неплохо сбросить эту информацию в файл, поэтому измените симулятор так, чтобы параллельно с выводом на экран он записывал копию данных в файл.

глава



Структуры данных



Цели

- Научиться динамически выделять и освобождать память для структур данных.
- Научиться организовывать связанные структуры данных с помощью указателей, структур, ссылающихся на себя, и рекурсии.
- Познакомиться с созданием и использованием связанных списков, очередей, стеков и двоичных деревьев.
- Познакомиться с важнейшими случаями применения связанных структур данных.

Содержание

12.1. Введение
12.2. Структуры, ссылающиеся на себя
12.3. Динамическое распределение памяти
12.4. Связанные списки
12.5. Стеки
12.6. Очереди
12.7. Деревья

Резюме • Распространенные ошибки программирования • Хороший стиль программирования • Советы по повышению эффективности • Советы по переносимости программ • Упражнения для самоконтроля • Ответы на упражнения для самоконтроля • Упражнения

12.1. Введение

Мы уже изучали ранее *структуры данных* фиксированного размера, такие, как одномерные массивы, двумерные массивы и структуры. В этой главе будут рассмотрены *динамические структуры данных*, размер которых, может увеличиваться или уменьшаться при исполнении программы. *Связанный список* — это набор «выстроенных в ряд» элементов данных, в любом месте которого можно производить вставки и удаления. *Стеки*, играющие важную роль в компьютерах и операционных системах, допускают выполнение вставки и удаления только с одного конца стека — *сверху*. *Очереди* представляют собой линии ожидания; вставки производятся в конец (иначе называемый *хвостом*) очереди, а удаление — в начале (называемом еще *головой*) очереди. *Двоичные деревья* облегчают скоростной поиск и сортировку данных, эффективное устранение повторяющихся элементов данных, представление файловой структуры каталогов и компиляцию выражений в машинный язык. Каждая из этих структур данных имеет множество других интересных применений.

Мы обсудим каждый из основных типов структур данных и рассмотрим программы, которые создают эти структуры и работают с ними. В следующей части книги — введении в C++ и объектно-ориентированное программирование, изложенном в главах с 15 по 21, — будет изучаться абстракция данных. Это позволит нам создавать подобные структуры данных совершенно иным способом, предназначенным для создания программного обеспечения, которое значительно проще сопровождать и, что особенно важно, легко использовать повторно.

Эта глава потребует от читателя определенных усилий. Разбираемые в главе учебные программы довольно близки к реальным, при их написании использована значительная часть того материала, что вы изучили в предыдущих главах. Особенно сложными являются операции с указателями. Наверное, многие из программистов на вопрос: «Какая тема была для вас наиболее сложна при изучении С?» не задумываясь, ответят: «Операции с указателями». И все же очень полезно разобраться в программах, содержащихся в этой главе, так как они пригодятся вам в дальнейшем при изучении более углубленных курсов. Кроме того, глава содержит богатую коллекцию упражнений, в которых придается особое значение использованию структур данных.

Авторы искренне надеются, что вы попытаетесь реализовать большой проект, описанный в специальном разделе «Как самому написать компилятор». Для того чтобы запустить написанную вами программу на С, вы пользуетесь компилятором, транслирующим эту программу в машинный язык. В результате реализации проекта вы действительно построите свой собственный компилятор. Он будет считывать файл, состоящий из операторов на простом, но довольно мощном языке высокого уровня, напоминающим ранние версии популярного языка BASIC. Далее ваш компилятор будет транслировать эти операторы в инструкции Машинного языка Симплетрона — языка, который вы изучили в специальном разделе главы 7, «Как самому построить компьютер». И написанная вами ранее программа-симулятор Симплетрона будет выполнять SML-программы, получаемые в результате работы компилятора, который вы создадите своими руками! Этот проект даст вам прекрасную возможность применить на практике большинство тех знаний, что были получены при изучении этого курса. Специальный раздел шаг за шагом проведет вас от описания высокоуровневого языка к алгоритмам, которые понадобятся для преобразования каждого оператора языка высокого уровня в машинный код. И если вы получите удовлетворение, закончив эту работу, то можете попробовать ввести как в компилятор, так и в симулятор Симплетрона те улучшения, что предлагаются в упражнениях.

12.2. Структуры, ссылающиеся на себя

Структуры, ссылающиеся на себя, содержат в качестве элемента указатель, который ссылается на структуру того же типа. Например, определение

```
struct node {
   int data;
   struct node *nextPtr;
};
```

описывает тип **struct node**. Структура типа **struct node** состоит из двух элементов — целого **data** и указателя **nextPtr**. Элемент **nextPtr** указывает на структуру типа **struct node** — структуру того же самого типа, что и только что объявленная нами, отсюда и термин «структура, ссылающаяся на себя». Элемент **nextPtr** иногда называют *связкой*, т.е. **nextPtr** можно использовать для того, чтобы связать структуру типа **struct node** с другой структурой того же типа. Структуры, ссылающиеся на себя, могут связываться вместе для образования полезных структур данных, таких как списки, очереди, стеки и деревья. Рис. 12.1 иллюстрирует две структуры, ссылающиеся на себя, связанные друг с другом и образующие список. Заметьте, что в связывающем элементе второй структуры нарисована представляющая указатель **NULL** косая черта, чтобы

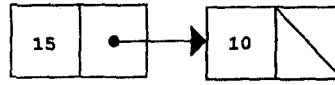


Рис. 12.1. Две ссылающихся на себя структуры, связанные друг с другом

показать, что этот элемент не указывает на другую структуру. Косая черта используется исключительно для иллюстрации и не имеет никакого отношения к символу обратной дробной черты языка С. Указатель **NULL** обычно обозначает конец структуры данных, так же как символ **NULL** обозначает конец строки.

Распространенная ошибка программирования 12.1

Связка последнего узла списка не устанавливается в **NULL**.

12.3. Динамическое распределение памяти

Создание и использование динамических структур данных требует *динамического распределения памяти — возможности получать в процессе* исполнения дополнительную память для хранения новых узлов и освобождать блоки памяти, ставшие ненужными. Максимальный размер выделяемой динамически памяти определяется доступной физической памятью компьютера или доступным виртуальным адресным пространством в системе с виртуальной памятью. Однако часто эти размеры значительно меньше, потому что память разделяется между многими пользователями (задачами).

Для динамического распределения памяти необходимо применение функций **malloc** и **free**, а также операции **sizeof**. Функция **malloc** принимает в качестве аргумента число байт, которое необходимо выделить, и возвращает указатель на выделенную память типа **void***. Указатель **void*** можно присвоить любой переменной-указателю. Функция **malloc** обычно используется совместно с операцией **sizeof**. Например, оператор

```
newPtr = malloc(sizeof(struct node));
```

оценивает **sizeof(struct node)** для определения размера в байтах структуры типа **struct node**, выделяет новый блок памяти размером в **sizeof(struct node)** байт, и сохраняет указатель на выделенную память в переменной **newPtr**. Если необходимого количества памяти нет в наличии, **malloc** возвращает указатель **NULL**.

Функция **free** освобождает память, т.е. память возвращается системе, и в дальнейшем ее можно выделить снова. Для высвобождения памяти, динамически выделенной предыдущим вызовом оператора **malloc**, используется оператор

```
free(newPtr);
```

В следующих разделах обсуждаются списки, стеки, очереди и деревья. Каждая из этих структур данных создается и используется с применением динамического распределения памяти и ссылающихся на себя структур.

Совет по переносимости программ 12.1

Размер структуры не обязательно равен сумме размеров ее элементов. Причина этого в существовании различных требований на выравнивание границ (см. главу 10).

Распространенная ошибка программирования 12.2

Считают, что размер структуры равен просто сумме размеров ее элементов.

Хороший стиль программирования 12.1

Используйте операцию **sizeof** для определения размера структуры.

Хороший стиль программирования 12.2

При использовании **malloc** проверяйте, не вернула ли функция указатель **NULL**. Напечатайте сообщение об ошибке, если запрашиваемая память не выделена.

Распространенная ошибка программирования 12.3

Если не освобождать выделенную динамически память после того, как она перестала использоваться, то это может привести к тому, что память в системе преждевременно исчерпается. Иногда это называют «утечкой памяти».

Хороший стиль программирования 12.3

Когда выделенная динамически память становится ненужной, вызывайте функцию **free**, чтобы немедленно вернуть память системе.

Распространенная ошибка программирования 12.4

Попытка освободить память, которая не была выделена динамически с помощью **malloc**.

Распространенная ошибка программирования 12.5

Ссылка на блок памяти, который был освобожден.

12.4. Связанные списки

Связанный список — это линейный набор ссылающихся на себя структур, называемых узлами, и объединенных указателем-*связкой*, отсюда и название — «связанный» список. Доступ к связанному списку обеспечивается указателем на первый узел списка. Доступ к следующим узлам производится через связывающий указатель, хранящийся в каждом узле. По общему соглашению связывающий указатель в последнем узле списка устанавливается в **NULL**, отмечая конец списка. Данные хранятся в связанном списке динамически — каждый узел создается по мере необходимости. Узел может содержать данные любого типа, включая другие структуры. Стеки и очереди также принадлежат к линейным структурам данных, и, как мы увидим, являются специальными вариантами связанных списков. Деревья же являются нелинейными структурами данных.

Списки данных могут храниться в массивах, однако связанные списки дают определенные преимущества. Связанный список удобен, когда заранее не известно, сколько элементов данных будет содержать структура. Связанные списки являются динамическими, поэтому длина списка при необходимо-

сти может увеличиваться или уменьшаться. Размер же массива изменить невозможно, потому что память под массив выделяется во время компиляции. Массив может переполниться. Связанные списки переполняются лишь в случае, если в системе не хватит памяти, чтобы удовлетворить очередной запрос на выделение динамической памяти.

Совет по повышению эффективности 12.1

Можно объявить массив заведомо большего размера, чем предполагаемое число элементов данных, однако это приводит к излишним затратам памяти. Связанные списки в подобной ситуации обеспечивают более рациональное использование памяти.

Связанные списки могут содержаться в сортированном виде, если помещать каждый новый элемент в соответствующую позицию списка.

Совет по повышению эффективности 12.2

Вставка и удаление в упорядоченном массиве требует определенного времени на выполнение, так как все элементы, следующие за вставляемым или удаляемым элементом, необходимо соответствующим образом сдвинуть.

Совет по повышению эффективности 12.3

Элементы массива хранятся в последовательных ячейках памяти. Это обеспечивает мгновенный доступ к любому элементу массива, поскольку адрес любого элемента можно найти исходя из его позиции относительно начала массива. Связанные списки не дают возможности подобного немедленного доступа к своим элементам.

Узлы связанных списков, как правило, не располагаются в памяти в виде одной сплошной области. Логически, однако, связанный список представляется непрерывным. Рис. 12.2 иллюстрирует связанный список с несколькими узлами.

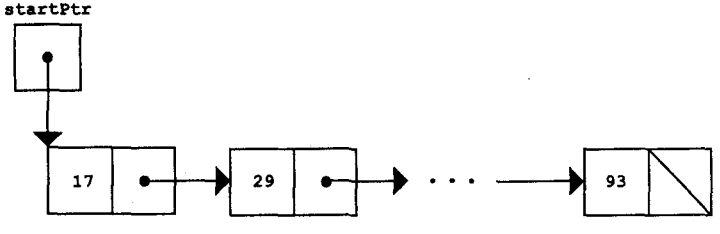


Рис. 12.2. *Графическое представление связанного списка*

Совет по повышению эффективности 12.4

Используя для структур данных, размер которых может увеличиваться и уменьшаться во время исполнения программы, динамическое распределение памяти (вместо массивов), можно сэкономить память. Не забывайте, однако, что указатели тоже занимают место в памяти и что динамическое выделение памяти влечет за собой дополнительные расходы времени на вызов функций.

Программа на рис. 12.3 (результат ее выполнения показан на рис. 12.4) управляет списком символов. Программа обеспечивает две возможности: 1) вставить символ в список в алфавитном порядке (функция **insert**) и 2) удалить символ из списка (функция **delete**). Это большая и сложная программа. Ниже мы детально ее обсудим. В упражнении 12.20 читателя просят написать рекурсивную функцию, выводящую на печать список в обратном порядке. А в упражнении 12.21 — рекурсивную функцию, осуществляющую поиск заданного элемента данных в связанном списке.

```
/* Операции над списком */
#include <stdio.h>
#include <stdlib.h>

struct listNode { /* структура со ссылкой на себя */
   char data;
   struct listNode *nextPtr;
};

typedef struct listNode LISTNODE;
typedef LISTNODE *LISTNODEPTR;

void insert(LISTNODEPTR *, char);
char delete(LISTNODEPTR *, char);
int isEmpty(LISTNODEPTR);
void printList(LISTNODEPTR);
void instructions(void);

main()
{
   LISTNODEPTR startPtr = NULL;
   int choice;
   char item;

   instructions(); /* вывести меню */
   printf("? ");
   scanf("%d", &choice);

   while (choice !=3) {

      switch (choice){
         case 1:
            printf("Enter a character: ");
            scanf("\n%c", &item);
            insert(&startPtr, item);
            printList(startPtr);
            break;
         case 2:
            if (!isEmpty(startPtr)) {
               printf("Enter character to be deleted: ");
               scanf("\n%c", &item);

               if (delete(&startPtr, item)) {
                  printf("%c deleted.\n", item);
                  printList(startPtr);
               }
```

Рис. 12.3. Вставка и удаление узлов в списке (часть 1 из 3)

```
            else
               printf("%c not found.\n\n",item);
         }
         else
            printf("List is empty.\n\n");

         break;
      default:
         printf("Invalid choice.\n\n");
         instructions();
         break;
      }

      printf("? ");
      scanf("%d", &choice);
   }

   printf("End of run.\n");
   return 0;
}

/* Распечатка инструкции */
void instructions(void)
{
   printf("Enter your choice:\n"
          " 1 to insert an element into the list.\n"
          " 2 to delete an element from the list.\n"
          " 3 to end.\n");
}

/* Вставка нового значения в упорядоченный список */
void insert(LISTNODEPTR *sPtr, char value)
{
   LISTNODEPTR newPtr, previousPtr, currentPtr;

   newPtr = malloc(sizeof(LISTNODE));

   if (newPtr != NULL) { /* доступна ли память? */
      newPtr->data = value;
      newPtr->nextPtr = NULL;

      previousPtr = NULL;
      currentPtr = *sPtr;

      while (currentPtr != NULL && value > currentPtr->data) {
         previousPtr = currentPtr;          /* перейти ...      */
         currentPtr = currentPtr->nextPtr; /* ... к следующему */
      }

      if (previousPtr == NULL) {
         newPtr->nextPtr = *sPtr;
         *sPtr = newPtr;
      }
      else {
         previousPtr->nextPtr = newPtr;
```

Рис. 12.3. Вставка и удаление узлов в списке (часть2 из 3)

```
            newPtr->nextPtr = currentPtr;
         }
      }
      else
         printf("%c not inserted. No memory available.\n", value);
}

/* Удаление элемента списка */
char delete(LISTNODEPTR *sPtr, char value)
{
   LISTNODEPTR previousPtr, currentPtr, tempPtr;

   if (value == (*sPtr)->data) {
      tempPtr = *sPtr;
      *sPtr = (*sPtr)->nextPtr; /* отсоединить узел */
      free(tempPtr);            /* освободить узел */
      return value;
   }
   else {
      previousPtr = *sPtr;
      currentPtr = (*sPtr)->nextPtr;

      while (currentPtr != NULL && currentPtr->data != value) {
         previousPtr = currentPtr;          /* перейти ...  */
         currentPtr = currentPtr->nextPtr; /* ... к следующему */
      }
      if (currentPtr != NULL) {
         tempPtr = currentPtr;
         previousPtr->nextPtr = currentPtr->nextPtr;
         free(tempPtr);
         return value;
      }
   }
   return '\0';
}

/* Возвратить 1, если список пуст, в противном случае - 0 */
int isEmpty(LISTNODEPTR sPtr)
{
   return sPtr == NULL;
}

/* Print the list */
void printList(LISTNODEPTR currentPtr)
{
   if (currentPtr == NULL)
      printf("List is empty.\n\n");
   else {
      printf("The list is:\n");

      while (currentPtr != NULL) {
         printf("%c--> ", currentPtr->data);
         currentPtr = currentPtr->nextPtr;
      }
      printf("NULL\n\n");
    }
}
```

Рис. 12.3. Вставка и удаление узлов в списке (часть 3 из 3)

```
Enter your choice:
   1 to insert an element into the list.
   2 to delete an element from the list.
   3 to end.
? 1
Enter a character:B
The list is:
B --> NULL

? 1
Enter a character: A
The list is:
A --> B --> NULL

? 1
Enter a character: C
The list is:
A --> B --> C --> NULL

? 2
Enter character to be deleted: D
D not found.

? 2
Enter character to be deleted: B
B deleted.
The list is:
A --> C --> NULL

? 2
Enter character to be deleted: C
C deleted.
The list is :
A --> NULL

? 2
Enter character to be deleted: A
A deleted.
List is empty.

? 4
Invalid choice.

Enter your choice:
   1 to insert an element into the list.
   2 to delete an element from the list.
   3 to end.
? 3
End of run.
```

Рис. 12.4. Пример результата работы программы на рис. 12.3

Две основных функции связанных списков — **insert** и **delete**. Функция **isEmpty** называется *предикатной функцией* — она никак не меняет список, а всего лишь определяет, является ли список пустым (т.е. указатель на первый узел списка равен **NULL**). Если список пуст, возвращается значение **1**, в противном случае **0**. Функция **printList** распечатывает список.

Символы вставляются в список в алфавитном порядке. Функции **insert** передаются *адрес* списка и символ, который необходимо вставить. Адрес списка необходим, когда значение должно быть вставлено в начало списка. Передача адреса списка позволяет модифицировать список (т.е. указатель на первый узел списка) через вызов по ссылке. Так как список сам по себе является указателем (на свой первый элемент), при передаче адреса списка создается *указатель на указатель* (другими словами, это *двойная косвенная адресация*). Это сложное понятие, требующее очень аккуратного программирования. Чтобы вставить символ в список, необходимо выполнить следующие шаги (см. рис. 12.5):

1) Вызвав **malloc**, создать узел, присвоив **newPtr** адрес выделенного блока памяти, присвоить **newPtr->data** символ, который необходимо вставить, а **newPtr->nextPtr** — значение **NULL**.

2) Инициализировать **previousPtr** как **NULL**, и **currentPtr** как ***sPtr** (указатель на начало списка). Указатели **previousPtr** и **currentPtr** используются для хранения положений узла, предшествующего точке вставки и узла после точки вставки.

3) Если **currentPtr** не **NULL**, и вставляемое значение больше, чем **currentPtr->data**, значение **currentPtr** присваивается **previousPtr**, а **currentPtr** перемещается в следующий узел списка. Так определяется точка в списке, в которую будет вставлено значение.

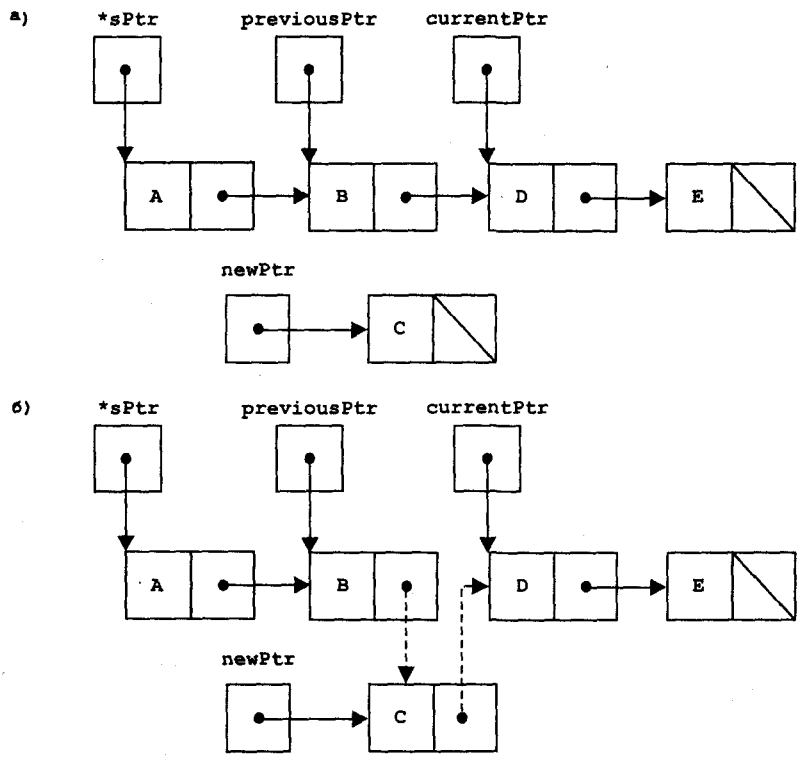


Рис. 12.5. Вставка узла на нужное место в списке

4) Если **previousPtr** равен **NULL**, новый узел вставляется как первый узел списка. Присвоив **newPtr->nextPtr** значение ***sPtr** (предыдущий первый узел — указателю-связке нового узла), и присвоив ***sPtr** значение **newPtr** (***sPtr** теперь указывает на новый узел). Если **previousPtr** не **NULL**, то новый узел устанавливается на свое место. Для этого значение **newPtr** присваивается **previousPtr->nextPtr** (указатель нового узла — связке предыдущего), а **newPtr->nextPtr** присваивается значение **currentPtr** (указатель текущего узла — связке нового).

Хороший стиль программирования 12.4

Присваивайте связующему элементу нового узла **NULL**. Указатели перед использованием должны быть инициализированы.

Рис. 12.5 иллюстрирует вставку узла, содержащего символ **'C'**, в упорядоченный список. Часть a) рисунка показывает список и новый узел перед вставкой. Часть b) — результат вставки нового узла. Модифицированные указатели показаны пунктиром.

Функция **delete** получает адрес указателя на начало списка и символ, который нужно удалить. Удаление символа из списка происходит следующим образом:

1) Если выясняется, что символ, который требуется удалить, находится в первом узле списка, **tempPtr** присваивается значение ***sPtr** (**tempPtr** будет использован, чтобы освободить ненужную более память), ***sPtr** присваивается значение **(*sPtr)->nexPtr** (***sPtr** теперь указывает на второй узел списка), блок памяти, на которую указывает **tempPtr**, освобождается, и функция возвращает символ, который был удален.

2) В противном случае происходит инициализация **previousPtr** значением ***sPtr** и **currentPtr** значением **(*sPtr)->nextPtr**.

3) Если **currentPtr** не **NULL** и значение, которое необходимо исключить, не равно **currentPtr->data**, **previousPtr** присваивается значение **currentPtr**, а **currentPtr** — значение **currentPtr->nexPtr**. Таким образом определяется местоположение символа, который требуется удалить, если он содержится в списке.

4) Если **currentPtr** не **NULL**, **tempPtr** присваивается значение **currentPtr**, **previousPtr->nextPtr** — значение **currentPtr->nextPtr**, узел, на который указывает **tempPtr**, освобождается, и возвращается символ, который был удален из списка. Если **currentPtr** равен **NULL**, возвращается **NULL** (символ **'\0'**) показывающий, что символ, который предлагается удалить, в списке не найден.

Рис 12.6 иллюстрирует удаление узла из связанного списка. Часть a) рисунка показывает связанный список после выполнения последней операции вставки. Часть b) показывает модификацию связывающего элемента **previousPtr** и присваивание **tempPtr** значения **currentPtr**. Указатель **tempPtr** используется для освобождения блока памяти, выделенного для хранения **'C'**.

Функция **printList** принимает в качестве аргумента указатель на начало списка и обращается к указателю **currentPtr**. Сначала функция определяет, не является ли список пустым. Если да, то **printList** выводит сообщение **"The list is empty."** и заканчивает работу. В противном случае она выводит данные

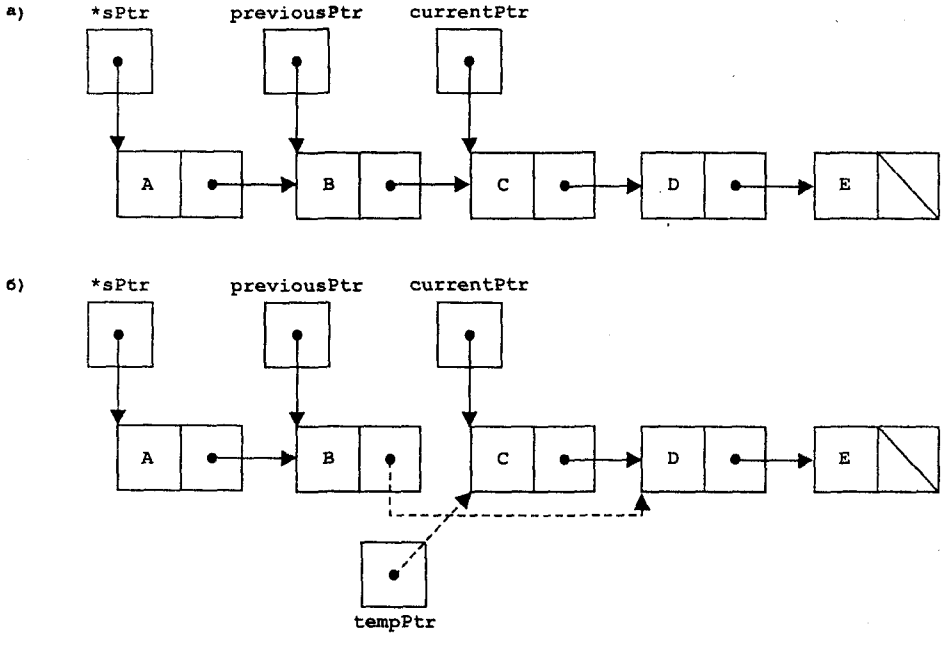


Рис. 12.6. Исключение узла из списка

списка. До тех пор, пока **currentPtr** не **NULL**, функция выводит на печать **currentPtr->data**, а **currentPtr** присваивает значение **currentPtr->nexPtr**. Заметим, что если связующий элемент в последнем узле не **NULL**, алгоритм печати будет выводить значения, находящиеся за границами списка, и это приведет к ошибке. Для связанных списков, стеков и очередей используется один и тот же алгоритм печати.

12.5. Стеки

Стек — это упрощенный вариант связанного списка. Новые узлы могут добавляться в стек и удаляться из стека только сверху. По этой причине, стек часто называют структурой вида *последним пришел — первым вышел* (LIFO). На стек ссылаются через указатель на верхний элемент стека. Связывающий элемент в последнем узле стека установлен равным **NULL**, чтобы пометить границу стека.

На рис. 12.7 показан стек с несколькими узлами. Заметим, что стеки и связанные списки представляются идентично. Разница между стеком и связанными списками в том, что вставку и удаление в связанных списках можно выполнять в любом месте, а в стеке только сверху.

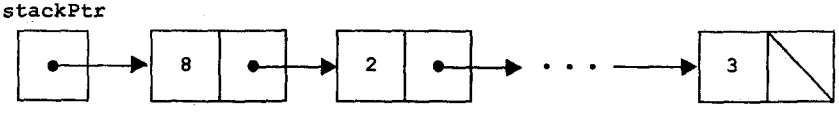


Рис. 12.7. Графическое представление стека

Распространенная ошибка программирования 12.6

Связке узла стека не присваивается **NULL**.

Основные функции, используемые при работе со стеками — **push** и **pop**. Функция **push** создает новый узел и помещает его на вершину стека. Функция **pop** удаляет верхний узел стека, освобождает память, которая была выделена изъятому узлу, и возвращает изъятое значение.

Программа на рис. 12.8 (результат ее выполнения представлен на рис. 12.9) работает с простым стеком целых чисел. Программа выполняет три действия на выбор: 1) помещает значение в стек (функция **push**), 2) изымает значение из стека (функция **pop**), и 3) завершает работу.

```
/* Программа динамического стека */
#include <stdio.h>
#include <stdlib.h>

struct stackNode { /* структура, ссылающаяся на себя */
   int data;
   struct stackNode *nextPtr;
};

typedef struct stackNode STACKNODE;
typedef STACKNODE *STACKNODEPTR;

void push(STACKNODEPTR *, int);
int pop(STACKNODEPTR *);
int isEmpty(STACKNODEPTR);
void printStack(STACKNODEPTR);
void instructions(void);

main()
{
   STACKNODEPTR stackPtr = NULL; /* указывает на вершину */
   int choice, value;

   instructions();
   printf("? ");
   scanf("%d", &choice) ;

   while (choice !=3) {

      switch (choice) {
         case 1:  /* заталкивает значение в стек */
            printf("Enter an integer: ");
            scanf("%d", &value);
            push(&stackPtr, value);
            printStack(stackPtr);
            break;
         case 2:  /* выталкивает значение из стека */
            if (!isEmpty(stackPtr))
               printf("The popped value is %d.\n",
                      pop(&stackPtr));
            printStack(stackPtr);
            break;
```

Рис. 12.8. Простая программа, иллюстрирующая работу со стеком (часть 1 из 3)

```
         default:
            printf("Invalid choice.\n\n");
            instructions();
            break;
      }

      printf("? ");
      scanf("%d", &choice);
   }

   printf("End of run.\n");
   return 0;
}

/* Распечатка инструкций */
void instructions(void)
{
   printf("Enter choice:\n"
          "1 to push a value on the stack\n"
          "2 to pop a value off the stack\n"
          "3 to end program\n");
}

/* Помещение нового значения на вершину стека */
void push (STACKNODEPTR *topPtr, int info)
{
   STACKNODEPTR newPtr;

   newPtr = malloc(sizeof(STACKNODE));
   if (newPtr != NULL) {
       newPtr->data = info;
       newPtr->nextPtr = *topPtr;
       *topPtr = newPtr;
   }
   else
      printf("%d not inserted. No memory available.\n", info);
}

/* Удаление узла на вершине стека */
int pop(STACKNODEPTR *topPtr)
{
   STACKNODEPTR tempPtr;
   int popValue;

    tempPtr = *topPtr;
    popValue = (*topPtr)->data;
    *topPtr = (*topPtr)->nextPtr;
    free(tempPtr);
    return popValue;
}

/* Распечатка стека */
void printStack(STACKNODEPTR currentPtr)
{
   if (currentPtr == NULL)
      printf("The stack is empty.\n\n");
```

Рис. 12.8. Простая программа, иллюстрирующая работу со стеком (часть 2 из 3)

```
   else {
      printf("The stack is:\n");

      while (currentPtr != NULL) {
         printf("%d ->", currentPtr->data);
         currentPtr = currentPtr->nextPtr;
      }

      printf("NULL\n\n");
   }
}

/* Стек пуст? */
int isEmpty(STACKNODEPTR topPtr)
{
   return topPtr == NULL;
}
```

Рис. 12.8. Простая программа, иллюстрирующая работу со стеком (часть 3 из 3)

```
Enter choice:
1 to push a value on the stack
2 to pop a value off the stack
3 to end program
? 1
Enter an integer: 5
The stack is:
5 --> NULL

? 1
Enter an integer :6
The stack is:
6 --> 5 --> NULL
? 1
Enter an integer: 4
The stack is:
4 --> 6 --> 5 --> NULL

? 2
The popped value is 4.
The Stack is:
6 --> 5 --> NULL

? 2
The popped value is 6.
The Stack is:
5 --> NULL

? 2
The popped value is 5.
The stack is empty.

? 2
The stack is empty.
```

Рис. 12.9. Результат выполнения программы на рис. 12.8 (часть 1 из 2)

```
? 4
Invalid choice.

Enter choice.:
1 to push a value on the stack
2 to pop a value off the stack
3 to end program
? 3
End of run.
```

Рис. 12.9. Результат выполнения программы на рис. 12.8 (часть 2 из 2)

Функция **push** помещает новый узел на вершину стека. В выполнении функции можно выделить три этапа:

1) Создание нового узла посредством вызова **malloc**, при этом указателю **newPtr** присваивается адрес блока выделенной памяти, переменной **newPtr->data** присваивается значение, которое необходимо поместить в стек, и указателю **newPtr->nextPtr** присваивается **NULL**.

2) Указателю **newPtr->nextPtr** присваивается ***topPtr** (указатель на вершину стека); связывающий элемент **newPtr** теперь указывает на узел, который был верхним до этого.

3) ***topPtr** присваивается значение **newPtr**; ***topPtr** теперь указывает на новую вершину стека.

Операции, включающие в себя ***topPtr**, меняют значение **stackPtr** в **main**. Рис. 12.10 дает наглядное представление о том, как происходит выполнение функции **push**. Часть a) рисунка показывает стек и новый узел перед операцией **push**. Пунктирные линии в части b) иллюстрируют шаги 2 и 3 выполнения **push**, в результате которых узел, содержащий **12**, становится новой вершиной стека.

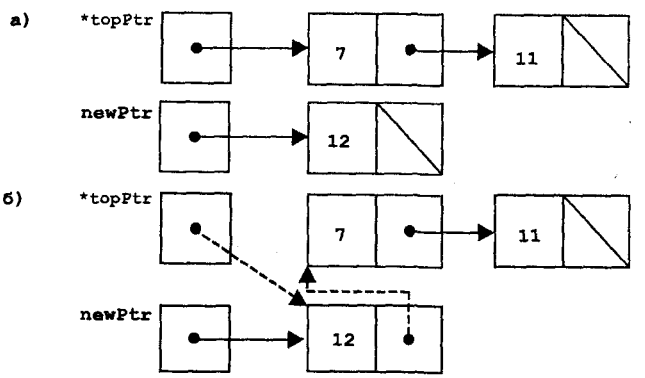


Рис. 12.10. Выполнение функции **push**

Функция **pop** удаляет верхний узел стека. Отметим, что перед тем как вызвать **pop**, **main** определяет, не пуст ли стек. В выполнении функции **pop** можно выделить пять основных этапов:

1) Указателю **tempPtr** присваивается ***topPtr** (**tempPtr** будет использован для освобождения ненужной памяти).

2) Переменной **popValue** присваивается значение **(*topPtr)->data** (сохраняется значение, находившееся в верхнем узле).

3) ***topPtr** присваивается **(*topPtr)->nextPtr** (***topPtr** присваивается адрес нового верхнего узла).

4) Освобождение памяти, на которую указывает **tempPtr**.

5) Вызывающей функции возвращается значение **popValue** (в программе на рис. 12.8. эта функция — **main**).

Рис. 12.11 иллюстрирует выполнение функции **pop**. Часть a) показывает стек после предыдущей операции **push**. В части b) выделены указатели **tempPtr**, указывающий на первый узел стека, и ***topPtr**, указывающий на второй узел. Для освобождения указанной **tempPtr** памяти вызывается функция **free**.

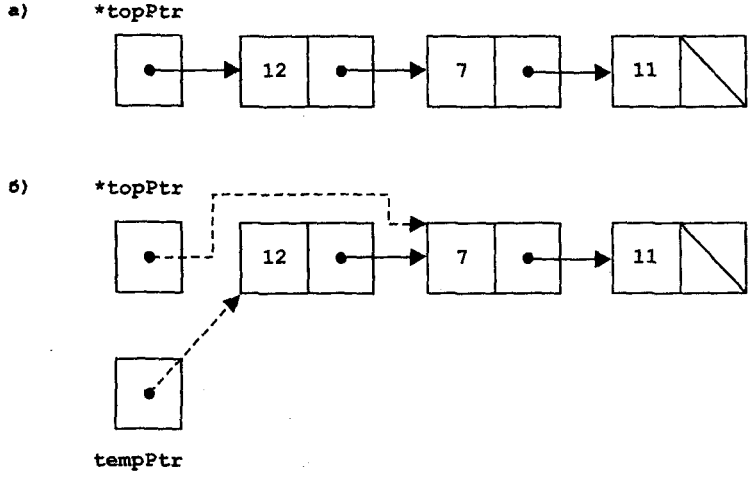


Рис. 12.11. Выполнение функции **pop**

Стеки имеют множество разнообразных применений. Например, всякий раз, когда происходит вызов функции, вызванная функция должна знать, как вернуться к вызывающей, поэтому адрес возврата помещается в стек. В случае, когда происходит целый ряд последовательных вызовов, адреса возврата размещаются в стеке в порядке последним пришел — первым вышел, так что после завершения выполнения каждой функции происходит переход к функции, ее вызывавшей. Стек поддерживает как обычные нерекурсивные вызовы, так и рекурсивный вызов функций.

На стеке выделяется пространство для автоматических переменных при каждом вызове функции. Когда происходит возврат от вызванной функции к вызывающей, пространство автоматических переменных функции удаляется из стека, и эти переменные становятся для программы неизвестными.

Компиляторы используют стеки в процессе вычисления выражений и создания кода машинного языка. В упражнениях будут исследованы некоторые применения стеков.

12.6. Очереди

Другой распространенной структурой данных является *очередь*. Очередь сходна с людьми у кассы в магазине: тот, кто ближе всех к кассе, обслуживается в первую очередь, другие покупатели пристраиваются в конец и ждут, пока их обслужат. В очереди узлы удаляются только с *головы*, а добавляются только в *хвост* очереди. По этой причине очереди часто называют структурами данных типа *первым пришел — первым ушел* (FIFO). Операции постановки в очередь и удаления из очереди носят названия *enqueue* (поставить в очередь) и *dequeue* (исключить из очереди).

Очереди находят в компьютерных системах многочисленные применения. Большинство компьютеров оборудовано только одним процессором, поэтому в каждый конкретный момент времени может обслуживаться только один пользователь (задача). Остальные же пользователи ставятся в очередь. Каждый стоящий в очереди постепенно перемещается к ее началу по мере обслуживания впереди стоящих пользователей.

Очереди также используются при организации буфера печати. Многопользовательская среда может иметь всего один принтер. Потребность вывести данные на принтер может возникнуть сразу у нескольких пользователей. Даже если принтер в этот момент занят одним из пользователей, остальные могут все же посылать свои данные на печать. Эти данные размещаются в буфере на диске, где и ожидают момента, когда получат доступ к принтеру.

Информационные пакеты в компьютерных сетях также проводят часть времени, ожидая в очередях. Пакет в сети переправляется от одного сетевого узла к другому, пока не достигнет узла назначения. Сетевой узел способен послать в каждый момент всего один пакет, поэтому все остальные пакеты становятся в очередь, ожидая, когда пересылающий узел сможет их обработать. Рис. 12.12 иллюстрирует очередь с несколькими узлами. Обратите внимание на указатели головы и хвоста очереди.

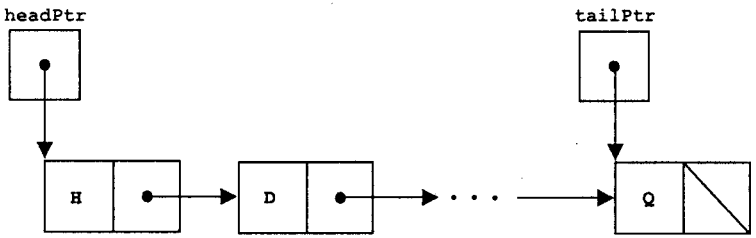


Рис. 12.12. Графическое представление очереди

Распространенная ошибка программирования 12.7

Связке последнего узла очереди не присваивается **NULL**.

Программа на рис. 12.13 (результат ее выполнения показан на рис. 12.14) — пример работы с очередью. Программа предлагает выполнить следующие действия на выбор: поставить узел в очередь (функция **enqueue**), удалить узел из очереди (функция **dequeue**), и выйти из программы.

```
/* Поддержка очереди */
#include <stdio.h>
#include <stdlib.h>

struct queueNode { /* структура, ссылающаяся на себя */
   char data;
   struct queueNode *nextPtr;
};

typedef struct queueNode QUEUENODE;
typedef QUEUENODE *QUEUENODEPTR;

/* function prototypes */
void printQueue(QUEUENODEPTR);
int isEmpty(QUEUENODEPTR);
char dequeue(QUEUENODEPTR *, QUEUENODEPTR *);
void enqueue(QUEUENODEPTR *, QUEUENODEPTR *, char);
void instructions(void);

main()
{
   QUEUENODEPTR headPtr = NULL, tailPtr = NULL;
   int choice;
   char item;

   instructions();
   printf("? ");
   scanf("%d", &choice);

   while (choice !=3) {

      switch(choice) {

         case 1:
            printf("Enter a character: ");
            scanf("\n%c", &item);
            enqueue(&headPtr, &tailPtr, item);
            printQueue(headPtr);
            break;
         case 2:
            if (!isEmpty(headPtr)) {
               item = dequeue(&headPtr, &tailPtr);
               printf("%c has been dequeued.\n", item);
            }

            printQueue(headPtr);
            break;

         default:
            printf("Invalid choice.\n\n");
            instructions();
            break;
      }

      printf("? ");
      scanf("%d", &choice);
   }
```

Рис. 12.13. Работа с очередью (часть 1 из 3)

```
   printf("End of run.\n");
   return 0;
}

void instructions(void)
{
   printf ("Enter your choice:\n"
           " 1 to add an item to the queue\n"
           " 2 to remove an item from the queue\n"
           " 3 to end\n");
}

void enqueue(QUEUENODEPTR *headPtr, QUEUENODEPTR *tailPtr,
             char value)
{
   QUEUENODEPTR newPtr;

   newPtr =malloc(sizeof(QUEUENODE));

   if (newPtr != NULL) {
      newPtr->data = value;
      newPtr->nextPtr = NULL;

         if (isEmpty(*headPtr))
            *headPtr = newPtr;
         else
            (*tailPtr)->nextPtr = newPtr;

         *tailPtr = newPtr;
      }
      else
         printf("%c not inserted. No memory available.\n", value);
   }

char dequeue(QUEUENODEPTR *headPtr, QUEUENODEPTR *tailPtr)
{
   char value;
   QUEUENODEPTR tempPtr;

   value = (*headPtr)->data;
   tempPtr = *headPtr;
   *headPtr = (*headPtr)->nextPtr;

   if (*headPtr == NULL)
      *tailPtr = NULL;

   free(tempPtr);
   return value;
}

int isEmpty(QUEUENODEPTR headPtr)
{
   return headPtr == NULL;
}
```

Рис. 12.13. Работа с очередью (часть 2 из 3)

```
void printQueue(QUEUENODEPTR currentPtr)
{
   if (currentPtr == NULL)
      printf("Queue is empty.\n\n");
   else {
      printf("The queue is ;\n");

      while (currentPtr != NULL) {
         printf("%c --> ", currentPtr->data);
         currentPtr = currentPtr->nextPtr;
      }

      printf("NULL\n\n");
   }
}
```

Рис. 12.13. Работа с очередью (часть 3 из 3)

```
Enter your choice:
   1 to add an item to the queue
   2 to remove an item from the queue
   3 to end
? 1
Enter a character: A
The queue is:
A --> NULL

? 1
Enter a character: B
The queue is:
A --> B --> NULL

? 1
Enter a character: C
The queue is:
A --> B --> C -->NULL

? 2
A has been dequeued.
The queue is:
B --> C --> NULL

? 2
B has been dequeued.
The queue is:
C --> NULL

? 2
C has been dequeued.
Queue is empty.

? 2
Queue is empty.

? 4
Invalid choice.
```

Рис. 12.14. Пример выполнения программы, представленной на рис. 12.13 (часть 1 из 2)

```
Enter your choice:
   1 to add an item to the queue
   2 to remove an item from the queue
   3 to end
? 3
End of run.
```

Рис. 12.14. Пример выполнения программы, представленной на рис. 12.13 (часть 2 из 2)

Функция **enqueue** получает от **main** три аргумента: адрес указателя на голову очереди, адрес указателя на хвост очереди и значение, которое необходимо поставить в очередь. Выполнение функции состоит из трех шагов:

1) Создание нового узла: вызвать **malloc**, присвоить адрес выделенного блока памяти **newPtr**, присвоить **newPtr->data** значение, которое необходимо поставить в очередь, а **newPtr->nextPtr** присвоить **NULL**.

2) Если очередь пуста, присвоить ***headPtr** указатель **newPtr**; в противном случае присвоить этот указатель **(*tailPtr)->nextPtr**.

3) И наконец, присвоить ***tailPtr** указатель **newPtr**.

Рис. 12.15 иллюстрирует выполнение функции **enqueue**. Часть a) рисунка показывает очередь и новый узел перед выполнением операции. Пунктирные линии в части b) показывают шаги 2 и 3 функции **enqueue**, которые позволяют добавить новый узел в конец очереди, которая не является пустой.

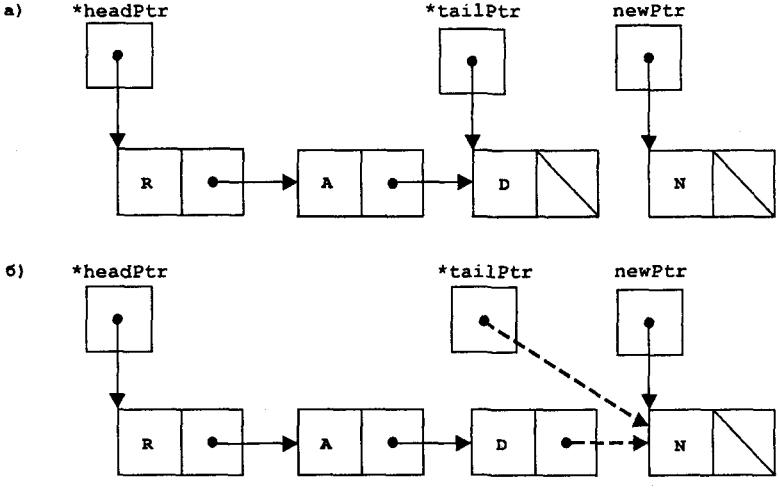


Рис. 12.15. Графическое представление выполнения функции **enqueue**

Функция **dequeue** получает в качестве аргументов адрес указателя на голову очереди и адрес указателя хвоста и удаляет первый узел очереди. Выполнение **dequeue** складывается из шести шагов:

1) Переменной **value** присвоить значение **(*headPtr)->data** (сохранить данные).

2) Присвоить указателю **tempPtr** значение ***headPtr** (**tempPtr** используется для освобождения ненужной памяти).

3) Присвоить указателю ***headPtr** значение **(*headPtr)->nextPtr**. (***headPtr** теперь указывает на новый первый узел очереди).

4) Если ***headPtr** указывает на **NULL**, установить ***tailPtr** также указывающим на **NULL**.

5) Освободить блок памяти, на который указывает **tempPtr**.

6) Передать значение **value** вызывающей функции (в программе на рис. 12.13 функция **dequeue** вызывалась из **main**).

Рис. 12.16 иллюстрирует выполнение функции **dequeue**. Часть a) показывает очередь после выполнения операции **enqueue**. Часть b) показывает **tempPtr**, указывающий на исключенный узел, и **headPtr**, указывающий на новый первый узел очереди. Для возвращения системе блока памяти, на который указывает **tempPtr**, вызывается функция **free**.

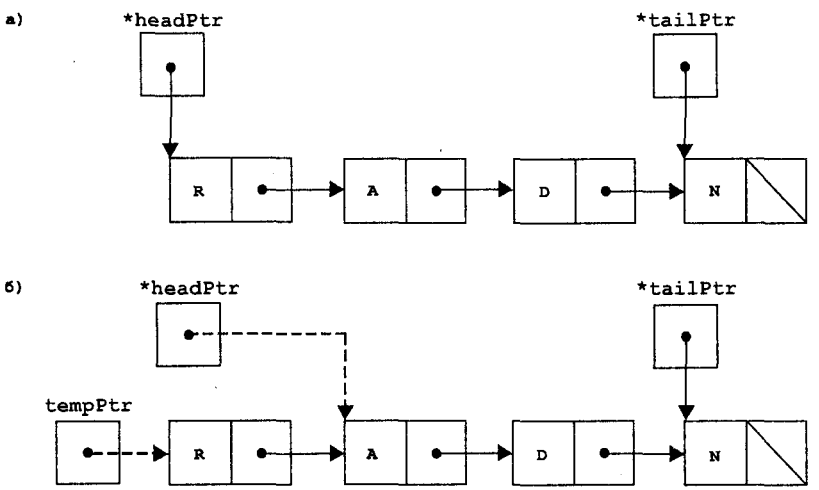


Рис. 12.16. Графическое представление выполнения функции **dequeue**

12.7. Деревья

Связанные списки, стеки и очереди — все это примеры *линейных структур данных*. Дерево же — нелинейная, двумерная структура данных с особыми свойствами. Узлы дерева содержат две или более связей. В этом разделе рассматриваются только *двоичные деревья* (рис. 12.17) — деревья, узлы которых содержат две связки (при этом как одна, так и обе связки могут быть **NULL**). Первый узел дерева называется *корневым*. Каждая связь корневого узла ссылается на *потомка*. *Левый потомок* — первый узел *левого поддерева*, а *правый потомок* —первый узел *правого поддерева*. Потомки одного узла называются *узлами-сиблингами*. Узел, не имеющий потомков, называется *листом*. Программисты обычно рисуют деревья от корневого узла вниз — т.е. в направлении, противоположном росту настоящих деревьев.

В этом разделе рассматривается специальное двоичное дерево, называемое *двоичным деревом поиска*. Двоичное дерево поиска (с неповторяющимися значениями в узлах) устроено так, что значения в любом левом поддереве меньше

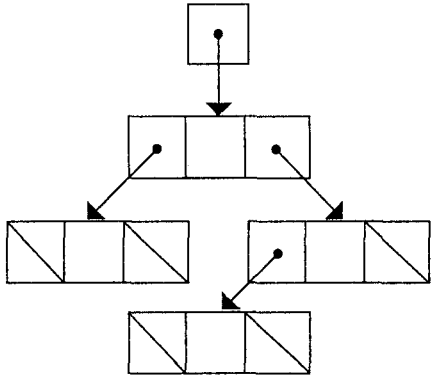


Рис. 12.17. Графическое представление двоичного дерева

ше, чем значение в родительском узле, а значения в любом правом поддереве больше, чем значение в родительском узле. На рис. 12.18 изображено двоичное дерево поиска с 12 значениями. Заметим, что форма двоичного дерева поиска для одного и того же набора данных может быть разной, в зависимости от порядка, в котором значения вставлялись в дерево.

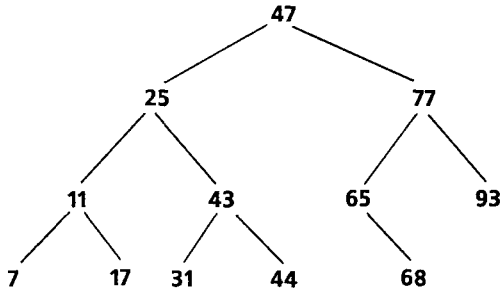


Рис. 12.18. Двоичное дерево поиска

Распространенная ошибка программирования 12.8

Связкам листовых узлов не присваивается **NULL**.

Программа на рис. 12.19 (результат ее выполнения представлен на рис. 12.20) создает двоичное дерево поиска и обходит его тремя способами: *с порядковой выборкой* (inorder), *предварительной выборкой* (preorder) и *отложенной выборкой* (postorder). Программа генерирует десять случайных чисел и вставляет их в дерево, за исключением повторяющихся, которые отбрасываются.

Функции, использованные в программе на рис. 12.19 для создания двоичного дерева поиска и его обхода, являются рекурсивными. Функции **insertNode** передаются два аргумента: адрес дерева и целое значение, которое необходимо сохранить в дереве. *В двоичном дереве поиска узел можно вставлять только в качестве листа.* Чтобы вставить узел в двоичное дерево поиска, необходимо выполнить следующее:

```
/* Создание двоичного дерева и его обход
   с порядковой, предварительной и отложенной выборкой */
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

struct treeNode {
   struct treeNode *leftPtr;
   int data;
   struct treeNode *rightPtr;
};

typedef struct treeNode TREENODE;
typedef TREENODE *TREENODEPTR;

void insertNode(TREENODEPTR *, int);
void inOrder(TREENODEPTR);
void preOrder(TREENODEPTR);
void postOrder(TREENODEPTR);

main()
{
   int i, item;
   TREENODEPTR rootPtr = NULL;

   srand(time(NULL));

   /* попытаться разместить в дереве 10 случайных чисел
      в диапазоне от 0 до 14 */
   printf("The numbers being placed in the tree are:\n");

   for (i = 1; i <= 10; i++) {
      item = rand() % 15;
      printf("%3d", item);
      insertNode(&rootPtr, item);
   }

   /* обойти дерево с предварительной выборкой */
   printf("\n\nThe preOrder traversal is:\n");
   preOrder(rootPtr);

   /* обойти дерево с порядковой выборкой */
   printf("\n\nThe inOrder traversal is:\n");
   inOrder(rootPtr);

   /* обойти дерево с отложенной выборкой */
   printf("\n\nThe postOrder traversal is:\n");
   postOrder(rootPtr);

   return 0;
}

void insertNode(TREENODEPTR *treePtr, int value)
{
   if (*treePtr == NULL) { /* *treePtr равен NULL */
      *treePtr = malloc(sizeof(TREENODE));
```

Рис. 12.19. Создание и обход двоичного дерева (часть 1 из 2)

```
      if (*treePtr != NULL) {
         (*treePtr)->data = value;
         (*treePtr)->leftPtr = NULL;
         (*treePtr)->rightPtr = NULL;
      }
      else
         printf("%d not inserted. No memory available.\n",
                value);
   }
   else
      if (value < (*treePtr)->data)
         insertNode(&((*treePtr)->leftPtr), value);
      else
         if (value > (*treePtr)->data)
            insertNode(&((*treePtr)->rightPtr), value);
         else
            printf("dup");
}

void inOrder(TREENODEPTR treePtr)
{
   if (treePtr != NULL) {
      inOrder(treePtr->leftPtr);
      printf("%3d", treePtr->data);
      inOrder(treePtr->rightPtr);
   }
}

void preOrder(TREENODEPTR treePtr)
{
   if (treePtr != NULL) {
      printf("%3d", treePtr->data);
      preOrder(treePtr->leftPtr);
      preOrder(treePtr->rightPtr);
   }
}

void postOrder(TREENODEPTR treePtr)
{
   if (treePtr != NULL) {
      postOrder(treePtr->leftPtr);
      postOrder(treePtr->rightPtr);
      printf("%3d", treePtr->data);
   }
}
```

Рис. 12.19. Создание и обход двоичного дерева (часть 2 из 2)

1) Если ***treePtr** равен **NULL**, создать новый узел. Вызвать **malloc**, присвоить выделенную память ***treePtr**, присвоить **(*treePtr)->data** целое значение, которое необходимо сохранить, присвоить **(*treePtr)->leftPtr** и **(*treePtr)->rightPtr** значение **NULL**, и вернуть управление вызывающей функции (либо **main**, либо предыдущему вызову **insertNode**).

2) Если значение ***treePtr** не **NULL** и значение, которое необходимо вставить меньше, чем **(*treePtr)->data**, функция **insertNode** вызывается с адресом **(*treePtr)->leftPtr**. В противном случае функция **insertNode**

вызывается с адресом **(*treePtr)->rightPtr**. Рекурсивные шаги продолжаются до тех пор, пока не будет обнаружен указатель **NULL**, после чего выполняется пункт 1) — вставка нового узла.

```
The numbers being placed in the tree are:
  7   8   0   6 14   1   0dup 13   0dup   7dup

The preOrder traversal is:
  7   0   6   1   8 14 13

The inOrder traversal is:
  0   1   6   7   8 13 14

The postOrder traversal is:
  1   6   0 13 14   8   7
```

Рис. 12.20. Пример выходных данных при выполнении программы рис. 12.19

Функции **inOrder**, **preOrder** и **postOrder** получают дерево (т.е. указатель на корневой узел) и проходят по нему.

В функции **inOrder** (порядковая выборка) выполняются следующие шаги:

1) Обойти с помощью **inOrder** левое поддерево.

2) Обработать значение в узле.

3) Обойти с помощью **inOrder** правое поддерево.

Значение в узле не обрабатывается до тех пор, пока не будут обработаны значения в его левом поддереве. Проход с помощью **inOrder** по дереву, изображенному на рис. 12.21, выглядит следующим образом:

```
6 13 17 27 33 42 48
```

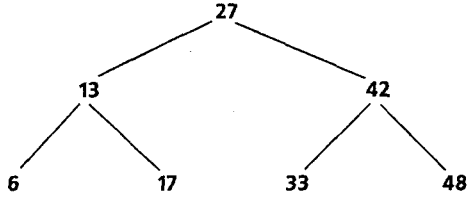


Рис. 12.21. Двоичное дерево поиска.

Отметим, что проход по двоичному дереву поиска с помощью **inOrder** выдает значения в узлах в возрастающем порядке. Процесс создания двоичного дерева поиска фактически сортирует данные — поэтому он называется *сортировкой двоичного дерева*.

В функции **preOrder** (предварительная выборка) выполняются следующие шаги:

1) Обработать значение в узле.

2) Обойти с помощью **preOrder** левое поддерево.

3) Обойти с помощью **preOrder** правое поддерево.

Значение в каждом узле обрабатывается при остановке в этом узле. После того, как значение узла обработано, обрабатываются значения в левом поддереве, а затем значения правого поддерева. Проход с помощью **preOrder** по дереву, изображенному на рис. 12.21, выглядит следующим образом:

```
27 13 6 17 42 33 48
```

В функции **postOrder** (отложенная выборка) выполняются следующие шаги:

1) Обойти с помощью **postOrder** левое поддерево.

2) Обойти с помощью **postOrder** правое поддерево.

3) Обработать значение в узле.

Значение в каждом узле не распечатывается до тех пор, пока не будут распечатаны значения его потомков. Проход с помощью **postOrder** по дереву, изображенному на рис. 12.21, выглядит следующим образом:

```
6 17 13 33 48 42 27
```

Двоичное дерево поиска упрощает удаление повторяющихся значений. Когда дерево создано, попытка вставить в него повторяющееся значение будет обнаружена, поскольку это значение будет следовать тем же самым решениям «поверни налево» или «поверни направо» при каждом сравнении, как и значение, введенное первым. Таким образом, повторяющееся значение будет в конце концов сравниваться с узлом, содержащим первое значение. И в этом месте его можно просто отбросить.

Также очень быстро можно провести в двоичном дереве поиск значения, совпадающего с заданным. Если дерево упаковано плотно, то каждый уровень содержит примерно в два раза больше элементов, чем предыдущий. Поэтому двоичное дерево поиска с n элементами должно содержать максимум $\log_2 n$ уровней и, таким образом, необходимо будет сделать максимум $\log_2 n$ сравнений, чтобы найти нужное значение, или же определить, что такого значения нет. Это означает, что когда, например, производится поиск в 1000-элемнтном двоичном дереве поиска (плотно упакованном), то необходимо сделать не более десяти сравнений, так как $2^{10} > 1000$. Для поиска же в двоичном дереве (плотно упакованном), содержащем 1000000 элементов, потребуется не больше двадцати сравнений, поскольку $2^{20} > 1000000$.

В упражнениях этой главы описаны алгоритмы для некоторых других операций с двоичными деревьями, таких, как удаление элемента из двоичного дерева, вывод двоичного дерева на печать в двумерном формате и выполнение обхода дерева по уровням. При проходе двоичного дерева по уровням узлы дерева обрабатываются ряд за рядом, начиная с уровня корневого узла. На каждом уровне узлы обрабатываются слева направо. В других упражнениях, посвященных двоичным деревьям, рассматривается еще целый ряд интересных вопросов: двоичное дерево поиска, которое может содержать одинаковые значения, размещение в двоичном дереве значений строкового типа, определение количества уровней, содержащихся в двоичном дереве.

Резюме

- Структуры, ссылающиеся на себя, содержат элемент-указатель, который ссылается на структуру того же типа.
- Структуры, ссылающиеся на себя, позволяют связывать между собой ряд однотипных структур, объединяя их в стеки, очереди, списки и деревья.

- Выделение динамической памяти резервирует определенное количество байт для хранения объекта во время выполнения программы.
- Функция **malloc** принимает в качестве аргумента число байт, которое необходимо выделить, и возвращает указатель типа **void** на выделенную память. Функция **malloc** обычно используется совместно с операцией **sizeof**. Операция **sizeof** возвращает размер в байтах для структуры, которой будет выделяться память.
- Функция **free** осуществляет освобождение памяти.
- Связанный список — это набор данных, хранящихся в группе связанных ссылающихся на себя структур.
- Связанный список — это динамическая структура данных; длина списка при необходимости может увеличиваться или уменьшаться.
- Размер связанного списка может увеличиваться до тех пор, пока имеется доступная память.
- Связанные списки обеспечивают простой механизм вставки и удаления данных путем модификации указателей.
- Стеки и очереди — это специальные разновидности связанного списка.
- Узлы добавляются в стек и удаляются из него только сверху. По этой причине стек называют структурой данных типа последним пришел — первым вышел.
- Связывающий элемент в последнем узле стека устанавливается равным **NULL**, чтобы отметить нижнюю границу стека.
- Для работы со стеками используются две основные операции **push** и **pop**. Операция **push** создает новый узел и помещает его на вершину стека. Операция **pop** удаляет узел с вершины стека, освобождает память, которая была выделена удаленному узлу, и возвращает удаляемое значение.
- В структуре данных типа очереди узлы удаляются с головы и добавляются в хвост. По этой причине очереди называют структурами данных типа первым пришел — первым вышел. Для добавления и удаления используют операции **enqueue** и **dequeue**.
- Деревья являются более сложными структурами данных, чем связанные списки, очереди и стеки. Деревья — это двумерные структуры данных, требующие двух или более связок на узел.
- Двоичные деревья содержат две связки на узел.
- Корневой узел является первым узлом дерева.
- Каждый из указателей в корневом узле ссылается на потомка. Левый потомок — первый узел в левом поддереве, а правый потомок — первый узел в правом поддереве. Потомки одного узла называются узлами-сиблингами. Если узел не имеет потомков, он называется листом.
- Двоичное дерево поиска устроено так, что значение в левом потомке меньше, чем в родительском узле, а значение в правом потомке больше или равно значению в родительском узле. Если повторяющиеся значения недопустимы, то значение в правом потомке просто больше значения в родительском узле.

- При выполнении прохода по двоичному дереву с порядковой выборкой сначала выполняется такой проход для левого поддерева, обрабатывается значение узла, после чего выполняется такой же проход правого поддерева. Значение в узле не обрабатывается, пока не обработаны все значения в левом поддереве.
- При выполнении прохода по двоичному дереву с предварительной выборкай сначала обрабатывается значение узла, затем выполняется проход левого поддерева и проход правого поддерева с предварительной выборкой. Значение в каждом узле обрабатывается сразу, как только он встречается.
- При выполнении прохода по двоичному дереву с отложенной выборкой сначала таким же образом проходится левое поддерево, затем правое и только после этого обрабатывается значение в узле. Значение в каждом узле не обрабатывается, пока не обработаны значения обоих его поддеревьев.

Терминология

dequeue
enqueue
FIFO (первым пришел — первым ушел)
free
LIFO (последним пришел — первым ушел)
malloc (выделить память)
pop
push
sizeof
вершина
вставка узла
голова очереди
двоичное дерево
двоичное дерево поиска
двойная косвенная адресация
дерево
динамические структуры данных
динамическое распределение памяти
корневой узел
левое поддерево
левый потомок
линейная структура данных
листовой узел
нелинейная структура данных
обход
обход с отложенной выборкой
обход с порядковой выборкой
обход с предварительной выборкой
остановка в узле
очередь
поддерево
потомки
правое поддерево
правый потомок
предикатная функция
родительский узел
связанный список
сиблинги
сортировка двоичного дерева
стек
структура, ссылающаяся на себя
удаление узла
узел
узел-потомок
указатель **NULL**
указатель на указатель
хвост очереди

Распространенные ошибки программирования

12.1. Связка последнего узла списка не устанавливается в **NULL**.

12.2. Считают, что размер структуры равен просто сумме размеров ее элементов.

12.3. Если не освобождать выделенную динамически память, после того, как она перестала использоваться, то это может привести к тому,

что память в системе преждевременно исчерпается. Иногда это называют «утечкой памяти».

12.4. Попытка освободить память, которая не была выделена динамически с помощью **malloc**.

12.5. Ссылка на блок памяти, который был освобожден.

12.6. Связке нижнего узла стека не присваивается **NULL**.

12.7. Связке последнего узла очереди не присваивается **NULL**.

12.8. Связкам листовых узлов не присваивается **NULL**.

Хороший стиль программирования

12.1. Используйте операцию **sizeof** для определения размера структуры.

12.2. При использовании **malloc** проверяйте, не вернула ли функция указатель **NULL**. Напечатайте сообщение об ошибке, если запрашиваемая память не выделена.

12.3. Когда выделенная динамически память становиться ненужной, используйте функцию **free**, чтобы немедленно вернуть память системе.

12.4. Присваивайте связующему элементу нового узла **NULL**. Указатели перед использованием должны быть инициализированы.

Советы по повышению эффективности

12.1. Можно объявить массив заведомо большего размера, чем предполагаемое число элементов данных, однако это приводит к излишним затратам памяти. Связанные списки в подобной ситуации обеспечивают более рациональное использование памяти.

12.2. Вставка и удаление в упорядоченном массиве требует определенного времени на выполнение, так как все элементы, следующие за вставляемым или удаляемым элементом, необходимо соответствующим образом сдвинуть.

12.3. Элементы массива хранятся в последовательных ячейках памяти. Это обеспечивает мгновенный доступ к любому элементу массива, поскольку адрес любого элемента можно найти исходя из его позиции относительно начала массива. Связанные списки не дают возможности подобного немедленного доступа к своим элементам.

12.4. Используя для структур данных, размер которых может увеличиваться и уменьшаться во время исполнения программы, динамическое распределение памяти (вместо массивов), можно сэкономить память. Не забывайте, однако, что указатели тоже занимают место в памяти и что динамическое выделение памяти влечет за собой дополнительные расходы времени на вызов функций.

Советы по переносимости программ

12.1. Размер структуры не обязательно равен сумме размеров ее элементов. Причина этого в существовании различных требований на выравнивание границ (см. главу 10).

Упражнения для самоконтроля

12.1. Заполните пробелы в каждом из следующих предложений:

a) _____ на себя структура используется для организации динамических структур данных.

b) Функция _____ используется для динамического выделения памяти.

c) _____ — редуцированный вариант связанного списка, в котором узлы могут вставляться и удаляться только в начале.

d) Функции, которые не изменяют связанные списки, а просто просматривают их, называют _____.

e) Очереди называют структурами данных типа _____, поскольку первый вставленный узел будет первым же и удален.

f) Указатель на следующий узел связанного списка называется _____.

g) Функция _____ используется для освобождения динамически выделенной памяти.

h) _____ — редуцированный вариант связанного списка, в котором узлы могут вставляться только в начало списка, а удаляться только из его конца.

i) _____ — нелинейная двумерная структура данных, содержащая узлы с двумя или более связями.

j) Стек называют структурой данных типа _____, потому что последний вставленный узел будет удален первым.

k) Узлы _____ дерева содержат по два связующих элемента.

l) Первый узел дерева является _____ узлом.

m) Каждая связь в узле дерева указывает на _____ или _____ этого узла.

n) Узел дерева, не имеющий потомков, называется _____.

o) Для прохождения двоичного дерева используются следующие три алгоритма обхода: _____, _____ и _____.

12.2. В чем заключается различие между связанным списком и стеком?

12.3. В чем заключается различие между стеком и очередью?

12.4. Напишите оператор или несколько операторов, выполняющие следующие задания. Исходите из того, что все действия выполняются в **main**, и сделаны следующие определения:

```
struct gradeNode {
   char lastName[20];
   float grade;
```

```
   struct gradeNode *nextPtr;
};
typedef struct gradeNode GRADENODE;
typedef GRADENODE *GRADENODEPTR;
```

a) Создайте указатель на начало списка с названием **startPtr**. Список пустой.

b) Создайте новый узел типа **GRADENODE**, на который указывает **newPtr** типа **GRADENODEPTR**. Присвойте строку **"Jones"** элементу **lastName** и значение **91.5** элементу **grade** (используйте **strcpy**). Напишите все необходимые объявления и операторы.

c) Предположим, что список, на который указывает **startPtr**, в настоящий момент состоит из двух узлов, один из которых содержит **"Jones"**, а второй **"Smith"**. Узлы расположены в алфавитном порядке. Напишите операторы, необходимые для того, чтобы вставить (с сохранением порядка) узлы, содержащие следующие данные для **lastName** и **grade**:

```
"Adams"      85.0
"Thompson"   73.5
"Pritchard"  66.5
```

Для выполнения вставок воспользуйтесь указателями **previousPtr**, **currentPtr** и **newPtr**. Выясните, на что указывают **previousPtr** и **currentPtr** перед каждой вставкой. Считается, что **newPtr** всегда указывает на новый узел, и что новому узлу уже были присвоены данные.

d) Напишите цикл **while**, который выводит на печать данные, находящиеся в каждом узле списка. Для перемещения по списку используйте указатель **currentPtr**.

e) Напишите цикл **while**, который удаляет все узлы списка и освобождает распределенные под узлы блоки памяти. Используйте указатель **currentPtr** и указатель **tempPtr** для перемещения вдоль списка и освобождения памяти соответственно.

12.5. Выполните вручную проходы по двоичному дереву поиска, изображенному на рис. 12.22, с порядковой, предварительной и отложенной выборкой.

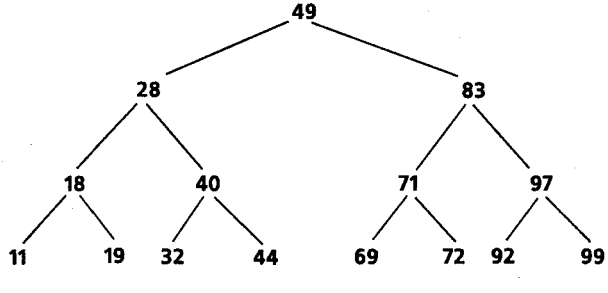


Рис. 12.22. Двоичное дерево поиска с 15-ю узлами

Ответы на упражнения для самоконтроля

12.1. a) ссылающаяся, b) **malloc**, c) Стек, d) предикатами, e) первым пришел — первым вышел, f) связкой, g) **free**, h) Очередь, i) Дерево, j) последним пришел, первым вышел, k) двоичного, l) корневым, m) потомка, поддерево, n) листом, o) с порядковой выборкой, с предварительной выборкой, с отложенной выборкой.

12.2. В связанном списке узел можно вставить в любом месте, и точно так же можно удалить любой узел. В стеке же узлы могут вставляться и удаляться только сверху.

12.3. У очереди существуют указатели как на голову, так и на хвост, поэтому узлы можно вставлять в хвост, а удалять из головы. Стек имеет всего один указатель — на вершину стека, где и выполняются как вставка, так и удаление узлов.

12.4. a)
```
GRADENODEPTR startPtr = NULL;
```

b)
```
GRADENODEPTR newPtr;
newPtr = malloc(sizeof(GRADENODE));
strcpy(newPtr->lastName, "Jones");
newPtr->grade = 91.5;
newPtr->nexPtr = NULL;
```

c) Вставка **"Adams"**:
previousPtr равен **NULL**, **currentPtr** указывает на первый элемент списка.

```
newPtr->nextPtr = currentPtr;
starPtr = newPtr;
```

Вставка **"Thompson"**:
previousPtr указывает на последний элемент списка (содержащий **"Smith"**)
currentPtr равен **NULL**

```
newPtr->nextPtr = currentPtr;
previousPtr->nextPtr = newPtr;
```

Вставка **"Pritchard"**:
previousPtr указывает на узел, содержащий **"Jones"**
currentPtr указывает на узел, содержащий **"Smith"**.

```
newPtr->nextPtr = currentPtr;
previousPtr->nexPtr = newPtr;
```

d)
```
currentPtr = startPtr;
while (currentPtr != NULL) {
   printf("Lastname = %s\nGrade = %6.2f\n",
          currentPtr->lastName, currentPtr->grade);
   currentPtr = currentPtr->nextPtr;
}
```

e)
```
currentPtr = startPtr;
while (currentPtr != NULL) {
   tempPtr = currentPtr;
   currentPtr = currentptr->nextPtr;
   free(tempPtr);
}
startPtr = NULL;
```

12.5. Обход с порядковой выборкой дает:

```
11 18 19 28 32 40 44 49 69 71 72 83 92 97 99
```

Обход с предварительной выборкой дает:

```
49 28 18 11 19 40 32 44 83 71 69 72 97 92 99
```

Обход с отложенной выборкой дает:

```
11 19 18 32 44 40 28 69 72 71 92 99 97 83 49
```

Упражнения

12.6. Напишите программу, выполняющую конкатенацию двух связанных списков символов. Программа должна включать функцию **concatenate**, которой в качестве аргументов передаются указатели на оба списка, и она присоединяет второй список к первому.

12.7. Напишите программу, которая объединяет два упорядоченных списка целых чисел в один упорядоченный список. Функция **merge** должна получать указатели на первый узел каждого списка, которые необходимо объединить, и вернуть указатель на первый узел объединенного списка.

12.8. Напишите программу, которая вставляет 25 случайных целых значений от 0 до 100 в упорядоченный связанный список. Программа должна вычислять сумму элементов и среднее значение, которое должно быть числом с плавающей точкой.

12.9. Напишите программу, которая создает связанный список из 10 символов, после чего создает копию списка с элементами, размещенными в обратном порядке.

12.10. Напишите программу, которая считывает строку текста и использует стек для того, чтобы распечатать эту строку в обратном порядке.

12.11. Напишите программу, которая использует стек для того, чтобы определить, является ли строка палиндромом (т.е. пишется по букам одинаково в прямом и в обратном направлении). Программа должна игнорировать пробелы и знаки пунктуации.

12.12. Стеки используются компиляторами как вспомогательный элемент в процессе оценки выражений и создания машинного кода. В этом и следующем упражнении мы выясним, как компиляторы оценивают арифметические выражения, состоящие только из констант, знаков операций и скобок.

Люди обычно записывают выражения в виде **3 + 4** и **7 / 9**, т.е. знак операции (в данном случае это + или /) записывается между операндами; такая запись называется *инфиксной нотацией*. Компьютеры «предпочитают» *постфиксную нотацию*, в которой знак операции записывается справа от двух операндов. Предыдущие инфиксные выражения в постфиксной нотации будут выглядеть соответственно как **3 4 +** и **7 9 /**.

Для оценки сложного инфиксного выражения компилятор должен сначала перевести его в постфиксную форму, а затем оценить постфиксный вариант выражения. Каждый из этих алгоритмов требует всего одного прохода выражения слева направо. Обоим алго-

ритмам требуется стек для выполнения операций, но стек в этих алгоритмах используется для разных целей.

В этом упражнении вы напишете реализацию на С алгоритма преобразования инфиксной записи в постфиксную. В следующем упражнении вы напишете реализацию алгоритма оценки постфиксных выражений.

Напишите программу, которая преобразует обычное инфиксное арифметическое выражение (подразумевается, что вводится допустимое выражение) с однозначными целыми, например, такое:

```
(6 + 2) * 5 - 8 / 4
```

в постфиксное выражение. Постфиксный вариант предыдущего инфиксного выражения выглядит следующим образом

```
6 2 + 5 * 8 4 / -
```

Программа должна считывать выражение в массив символов **infix** и использовать видоизмененные варианты функций работы со стеком, представленных в этой главе, чтобы с их помощью сохранить постфиксное выражение в символьном массиве **postfix**. Алгоритм создания постфиксного выражения следующий:

1) Ввести в стек левую скобку '('.
2) Добавить правую скобку ')' в конец **infix**.
3) Пока стек не пуст, считывать **infix** слева направо и выполнять следующие действия:

Если текущий символ в **infix** — цифра, копируйте его в следующий элемент **postfix**.

Если текущий символ в **infix** — левая скобка, помещайте ее в стек.

Если текущий символ в **infix** — знак операции,

> извлекайте знаки операций из стека (если они там есть), пока соответствующие им операции имеют равный или более высокий приоритет по сравнению с текущей операцией, и вставляйте извлеченные знаки операций в **postfix**.

Вставьте текущий символ из **infix** в стек.

Если текущий символ в **infix** — правая скобка,

> извлекайте знаки операций из стека и вставляйте их в **postfix** до тех пор, пока на вершине стека не появится левая скобка.

Извлеките из стека левую скобку и отбросьте ее.

В выражении допускается использование следующих арифметических операций:

```
+   сложение
-   вычитание
*   умножение
/   деление
^   возведение в степень
%   взятие по модулю
```

Стек задается с помощью следующих объявлений:

```
struct stackNode {
   char data;
   struct stackNode *nextPtr;
};
```

```
typedef struct stackNode STACKNODE;
typedef STACKNODE *STACKNODEPTR;
```

Программа должна состоять из функции **main** и восьми других функций со следующими заголовками:

```
void convertToPostfix(char infix[], char postfix[])
```

Преобразует инфиксное выражение в постфиксную нотацию.

```
int isOperator(char c)
```

Определяет, является ли **с** знаком операции.

```
int precedence(char operator1, char operator2)
```

Определяет, что старшинство операции **operator1** меньше, равно или выше по сравнению с **operator2**. *Функция возвращает* соответственно **-1**, **0** или **1**.

```
void push(STACKNODEPTR *topPtr, char value)
```

Помещает значение в стек.

```
char pop(STACKNODEPTR *topPtr)
```

Извлекает значение из стека.

```
char stackTop(STACKNODEPTR topPtr)
```

Возвращает значение в верхнем узле стека, без извлечения этого значения.

```
int isEmpty(STACKNODEPTR topPtr
```

Определяет, является ли стек пустым.

```
void printStack(STACKNODEPTR topPtr)
```

Выводит стек на печать.

12.13. Напишите программу, оценивающую постфиксное выражение (предполагается, что вводится допустимое выражение), например такое:

```
6 2 + 5 * 8 4 / -
```

Программа должна вводить постфиксное выражение, состоящее из цифр и знаков операций, в символьный массив. Используя видоизмененные варианты функций работы со стеком, рассмотренных ранее в этой главе, программа должна считывать выражение и оценивать его. Используйте следующий алгоритм:

1) Добавить символ **NULL** (**'\0'**) в конец постфиксного выражения. *Как только встречается* символ **NULL**, дальнейшую обработку можно прекратить.

2) Пока не встретился **'\0'** , считывать выражение слева направо.
Если текущий символ — цифра,
поместить ее значение в стек (значение цифры равно значению ее символа в таблице символов компьютера минус значение символа **'0'**).
В противном случае, если текущий символ — знак операции,
извлечь два верхних элемента из стека в переменные **x** и **y**.
Произвести вычисление **y** *операция* **x**.
Поместить результат вычисления в стек.

3) Когда в выражении встретится символ **NULL**, извлечь из стека верхнее значение. Оно и будет результатом постфиксного выражения.

Замечание: в пункте 2), *если знак операции* — '/', наверху в стеке находится **2**, и следующий элемент в стеке **8**, то **2** помещается в **x**, **8** помещается в **y**, вычисляется **8 / 2**, и результат **4** заталкивается обратно в стек. Все вышесказанное в равной степени относится и к операции '-'. В выражении допустимы следующие арифметические операции:

```
+   сложение
-   вычитание
*   умножение
/   деление
^   возведение в степень
%   взятие по модулю
```

Стек задается с помощью следующих объявлений:

```
struct stackNode {
   int data;
   struct stackNode *nextPtr;
};

typedef struct stackNode STACKNODE;
typedef STACKNODE *STACKNODEPTR;
```

Программа должна состоять из функции **main** и восьми других функций со следующими заголовками:

```
int evaluatePostfixExpression(char *expr)
```

Вычисляет постфиксное выражение.

```
int calculate(int op1, int op2, char operator)
```

Вычисляет выражение **op1 operator op2**.

```
void push(STACKNODEPTR *topPtr, int value)
```

Помещает значение в стек.

```
int pop(STACKNODEPTR *topPtr)
```

Извлекает значение из стека.

```
int isEmpty(STACKNODEPTR topPtr)
```

Определяет, является ли стек пустым.

```
void printStack(STACKNODEPTR topPtr)
```

Выводит стек на печать.

12.14. Усовершенствуйте программу упражнения 12.13, вычисляющую постфиксное выражение, чтобы она могла работать с целыми операндами, большими девяти.

12.15. (*Моделирование супермаркета*) Напишите программу, которая моделирует очередь в кассу супермаркета. Покупатели подходят к кассе через случайно выбранные целые интервалы времени, заключенные в диапазоне от одной до четырех минут. На обслуживание каждого покупателя тратится тоже случайно выбранное целое число минут от одной до четырех. Очевидно, что время, через которое появляется новый покупатель, и время, затрачиваемое на обслужи-

вание покупателя, должны быть уравновешены. Если в среднем частота появления покупателей у кассы больше, чем среднее время обслуживания, очередь будет бесконечно расти. Даже когда эти две величины сбалансированы, из-за их вероятностного характера все равно могут образовываться длинные очереди. Создайте программу моделирования для 12-ти часового дня (720 минут), используя следующий алгоритм:

1) Случайным образом выбирается целое число от одного до четырех, для определения минуты, в которую появляется первый покупатель.
2) После появления первого покупателя:
Определите время обслуживания покупателя (случайно выбранное целое число от 1 до 4).
Начните обслуживание покупателя.
Найдите время появления следующего покупателя (к текущему времени прибавляется случайно выбранное целое число от 1 до 4).
3) Для каждой минуты рабочего дня супермаркета:
Если появился следующий покупатель,
поставьте его в очередь.
Определите время появления следующего покупателя.
Если обслуживание очередного покупателя завершено,
извлеките из очереди следующего покупателя, который должен обслуживаться.
Определите время обслуживания покупателя (случайно выбранное целое число от 1 до 4).

Теперь запустите ваше моделирование для 720 минут и ответьте на следующие вопросы:

a) Какое максимальное количество покупателей оказалось в очереди за это время?
b) Чему равно максимальное время, которое покупателю пришлось ждать обслуживания?
c) Что произойдет, если интервал появления покупателей поменять с от одной до четырех минут на от одной до трех минут?

12.16. Измените программу на рис. 12.19 так, чтобы двоичное дерево могло содержать повторяющиеся значения.

12.17. На основе программы, представленной на рис. 12.19, напишите программу, которая вводит строку текста, разделяет предложение на отдельные слова, вставляет слова в двоичное дерево поиска и распечатывает результаты его прохода тремя способами: с порядковой, предварительной и отложенной выборкой.

Подсказка: считайте строку текста в массив. Вызовите **strtok** для отделения текущего слова. Когда слово выделено, создайте новый узел дерева, присвойте указатель, возвращаемый **strtok**, элементу **string** нового узла, и вставьте узел в дерево.

12.18. В этой главе мы увидели, что при создании двоичного дерева поиска повторы устраняются очень просто. Опишите, как бы вы выполнили удаление повторов, если бы использовали просто одномерный массив переменных. Сравните удаление повторений из массива и ту же самую операцию в двоичном дереве поиска.

12.19. Напишите функцию **depth**, принимающую в качестве аргумента принимающую двоичное дерево и определяющую, сколько уровней оно содержит.

12.20. (*Рекурсивный вывод на печать списка в обратном порядке*) Напишите функцию **printListBackwards**, которая рекурсивно выводит элементы данных в списке в обратном порядке. Используйте вашу функцию в тестовой программе, которая создает упорядоченный список целых чисел и распечатывает его в обратном порядке.

12.21. (*Рекурсивный поиск в списке*) Напишите функцию **searchList**, которая выполняет рекурсивный поиск заданного значения в связанном списке. В том случае, если оно обнаружено, функция должна возвращать указатель на значение; в противном случае должно возвращаться значение **NULL**. Используйте вашу функцию в тестовой программе, которая создает список целых чисел. Программа должна предлагать пользователю ввести значение, которое необходимо обнаружить в списке.

12.22. (*Удаление двоичного дерева*) В этом упражнении мы обсудим удаление элементов данных из двоичного дерева поиска. Алгоритм удаления не так прост, как алгоритм вставки. При удалении элемента данных возможен один из трех случаев: элемент данных содержится в листе (т.е. узле, не имеющем потомков), в узле, имеющем одного потомка, или в узле, имеющем двух потомков.

Если удаляемый элемент содержится в листе, лист удаляется, а указателю в его родительском узле присваивается **NULL**.

Если удаляемый элемент принадлежит узлу с одним потомком, то указатель в его родительском узле устанавливается на потомка, а узел, содержащий элемент данных, удаляется. В этом случае потомок занимает в дереве место удаленного узла.

Третий случай наиболее сложен. Если удаляется узел, имеющий двух потомков, его место должен занять другой узел дерева. Однако нельзя просто присвоить указателю в родительском узле указатель на одного из потомков узла, который должен быть удален. В большинстве случаев получающееся в результате дерево не будет удовлетворять следующему критерию двоичного дерева поиска: *значения в любом левом поддереве должны быть меньше, чем значение в родительском узле, а значения в любом правом поддереве должны быть больше, чем значение в родительском узле.*

Какой же узел должен быть использован в качестве *замещающего узла*, чтобы удовлетворить этому условию? Ответ такой: или узел, содержащий наибольшее из всех значений в дереве, меньшее, чем значение в удаляемом узле, или узел, содержащий наименьшее из всех значений в дереве, большее, чем значение в удаляемом узле. Давайте рассмотрим узел с меньшим значением. В двоичном дереве поиска наибольшее значение, меньшее, чем родительское значение, расположено в левом поддереве родительского узла и гарантированно будет содержаться в самом правом узле поддерева. Этот узел можно найти, спускаясь вниз по левому поддереву вправо до тех пор, пока указатель на правого потомка в текущем узле не будет равен **NULL**. Это и есть интересующий нас узел, который будет

или листом, или узлом, имеющим только одного потомка, причем слева. Если замещающий узел — лист, то для удаления необходимо выполнить следующие шаги:

1) Сохранить указатель на узел, который необходимо удалить, во временном указателе (этот указатель будет использован для освобождения выделенной динамически памяти).
2) Установить указатель в родительском, по отношению к удаляемому, узле, на замещающий узел.
3) Установить указатель в родительском, по отношению к замещающему, узле на **NULL**.
4) Установить указатель на правое поддерево в замещающем узле на правое поддерево удаляемого узла.
5) Удалить узел, на который указывает временный указатель.

Шаги, которые необходимо выполнить при удалении в случае, если заменяющий узел имеет левого потомка, такие же, как и при замене узла, не имеющего потомков, но алгоритм, кроме этого, должен переместить потомка в позицию замещающего узла. Если замещающий узел — узел с левым потомком, то для удаления необходимо выполнить следующие шаги:

1) Сохранить указатель на узел, который должен быть удален, во временной переменной.
2) Установить указатель в родительском, по отношению удаляемому, узле на замещающий узел.
3) Установить указатель в родительском, по отношению к замещающему, узле на левого потомка замещающего узла.
4) Установить указатель на правое поддерево в замещающем узле на правое поддерево удаляемого узла.
5) Удалить узел, на который указывает временный указатель.

Напишите функцию **deleteNode**, которая принимает в качестве аргументов указатель на корневой узел дерева и значение, которое необходимо удалить. Функция должна находить в дереве узел, содержащий указанное значение, и применить обсуждавшийся выше алгоритм для удаления узла. Если требуемое значение в дереве не обнаружено, функция должна напечатать соответствующее сообщение. Модифицируйте программу на рис. 12.19 так, чтобы она вызывала эту функцию. После удаления элемента данных вызовите функции обхода **inOrder**, **preOrder** и **postOrder** для проверки корректности удаления.

12.23. (*Поиск в двоичном дереве*) Напишите функцию **binaryTreeSearch**, которая будет искать в двоичном дереве заданное значение. Функция должна принимать в качестве аргументов указатель на корневой узел двоичного дерева поиска и ключевое значение, которое необходимо найти. Если узел, содержащий ключевое значение, обнаружен, функция должна вернуть указатель на этот узел; в противном случае функция должна возвратить **NULL**.

12.24. (*Обход двоичного дерева по уровням*) Программа на рис. 12.19 иллюстрирует три рекурсивных метода обхода двоичных деревьев: с порядковой, предварительной и отложенной выборкой. В этом упражнении обход двоичного дерева производится *по уровням*, т.е.

значения узлов выводятся на печать ряд за рядом, начиная от корневого узла. Узлы каждого уровня печатаются слева направо. Обход по уровням не является рекурсивным алгоритмом. Он использует структуру данных типа очереди для управления выводом значений узлов. Алгоритм заключается в следующем:

1) Поместить корневой узел в очередь.
2) Пока в очереди остаются узлы,
   прочитать следующий узел в очереди
   распечатать значение в узле
   если указатель на левого потомка узла не **NULL**
     вставить левого потомка в очередь
   если указатель на правого потомка не **NULL**
     вставить правого потомка в очередь.

Напишите функцию **levelOrder**, выполняющую обход двоичного дерева по уровням. Функция должна принимать в качестве аргумента указатель на корневой узел двоичного дерева. Измените программу на рис. 12.19 так, чтобы можно было протестировать эту функцию. Сравните результаты вывода функции с результатами других алгоритмов обхода, чтобы убедиться, что она работает правильно. (Примечание: вам также придется изменить и включить в эту программу функцию, обрабатывающую очередь, представленную на рис. 12.13).

12.25. (*Вывод деревьев на печать*) Напишите рекурсивную функцию **outputTree** для изображения на экране двоичного дерева. Функция должна выводить дерево ряд за рядом, вершина дерева должна располагаться в левой части экрана, а его низ справа. Каждый ряд выводится вертикально. Например, изображение двоичного дерева на рис. 12.22 должно выглядеть на экране так:

```
                99
          97
                92
     83
                72
          71
                69
49
                44
          40
                32
     28
                19
          18
                11
```

Отметим, что крайний справа листовой узел появляется вверху в самой правой колонке, а корневой узел появляется в левой части экрана. Каждый столбец выводится через пять пробелов от предыдущего. Функция **outputTree** должна получать в качестве аргументов указатель на корневой узел дерева и целое число **totalSpaces**, задающее число пробелов перед очередным столбцом (эта переменная должна иметь начальное значение равное нулю, чтобы корневой узел выводился с левого края экрана). Функция использует мо-

дифицированный обход дерева с порядковой выборкой. Он начинается в крайнем правом узле дерева, от которого перемещается влево. Алгоритм выполняется следующим образом:

Пока указатель на текущий узел не **NULL**
рекурсивно вызовите **outputTree** с правым поддеревом текущего узла и **totalSpaces+5**
воспользуйтесь структурой **for** для подсчета от 1 до **totalSpaces** и вывода пробелов
выведите значение в текущем узле
установите указатель текущего узла на *левое поддерево* текущего узла
увеличьте значение **totalSpaces** на 5.

Специальный раздел: как самому написать компилятор

В упражнениях 7.18 и 7.19, мы ввели Машинный язык Симплетрона (SML) и создали симулятор компьютера, выполняющий программы, написанные на SML. В этом разделе мы создадим компилятор, который будет транслировать программы, написанные на языке программирования высокого уровня, в SML. В этом разделе «связываются» воедино отдельные моменты процесса программирования. Мы будем писать программы на языке высокого уровня, компилировать их с помощью компилятора, который мы здесь создадим, и запускать компилированные программы на симуляторе, созданном в упражнении 7.19.

12.26. (*Язык Simple*) Прежде чем мы начнем разрабатывать компилятор, мы обсудим простой, но в тоже время достаточно мощный, язык высокого уровня, похожий на первые версии популярного языка BASIC. Назовем его язык *Simple*. Каждый *оператор* состоит из *номера строки* и *инструкции* Simple. Номера строк должны следовать в возрастающем порядке. Каждая инструкция начинается с одной из следующих *команд* Simple: **rem**, **input**, **let**, **print**, **goto**, **if/goto** и **end** (см. рис. 12.23). Все команды, за исключением **end**, могут встречаться в программе многократно. Simple оперирует только с целыми выражениями, включающими операции +, -, * и /. Эти операции имеют такое же старшинство, как и аналогичные операции C. Для того, чтобы изменить порядок вычисления в выражении, можно использовать скобки.

Наш компилятор Simple распознает только буквы нижнего регистра. Все символы в файле Simple должны быть набраны в нижнем регистре (буквы, набранные в верхнем регистре, приведут к синтаксической ошибке, если только они не входят в оператор **rem**, для которого регистр игнорируется). *Имя переменной* состоит из одной буквы. Simple не позволяет использовать осмысленные имена переменных, поэтому их желательно описывать в комментариях. Simple оперирует только с целыми переменными. В нем отсутствует объявление переменных — простое упоминание имени переменной в программе автоматически вызывает ее объявление и присвоение нулевого значения. Синтаксис языка Simple не позволяет выполнять операций со строками (чтение строки, запись стро-

ки, сравнение двух строк и т.д.). Если в программе на Simple встречается строка (после команды, отличной от **rem**), компилятор генерирует сообщение о синтаксической ошибке. В нашем компиляторе предполагается, что программа на Simple введена правильно. В упражнении 12.29 читателю будет предложено усовершенствовать компилятор, чтобы он мог выполнять проверку программы на синтаксические ошибки.

Команды	Примеры операторов	Описания
rem	50 rem this is a remark	Любой текст после команды **rem** используется исключительно для целей документирования и игнорируется компилятором.
input	30 input x	Выводит на экран знак вопроса, предлагая пользователю ввести целое число. Считывает число, введенное с клавиатуры, и сохраняет в **x**.
let	80 let u = 4 * (j - 56)	Присваивает **u** значение выражения **4 * (j - 56)**. Отметим, что справа от знака равенства может стоять выражение любой степени сложности.
print	10 print w	Выводит на экран значение **w**
goto	70 goto 45	Передает управление строке **45**.
if/goto	35 if i == z goto 80	Сравнивает **i** и **z** на предмет равенства и передает управление строке **80**, если условие истинно; в противном случае выполняется следующий за **if/goto** оператор.
end	99 end	Прерывает исполнение программы.

Рис. 12.23. Команды Simple

Simple использует операторы условного (**if/goto**) и безусловного (**goto**) переходов для изменения порядка выполнения операций в процессе выполнения программы. Если условие в операторе **if/goto** истинно, управление передается заданной в операторе строке программы. В операторе **if/goto** допустимы следующие операции отношений: <, >, <=, >=, == или !=. Старшинство этих операций такое же, как в языке C.

Давайте рассмотрим теперь несколько программ на языке Simple, демонстрирующих его возможности. Первая программа (на рис. 12.24) считывает два целых числа, введенных с клавиатуры, сохраняет их значения в переменных **a** и **b**, после чего вычисляет и выводит на печать их сумму (хранящуюся в переменной **c**).

Программа на рис. 12.25 определяет и выводит на печать большее из двух целых чисел. Числа вводятся с клавиатуры и сохраняются в переменных **s** и **t**. Оператор **if/goto** проверяет на истинность условие **s >= t**. Если оно истинно, управление предается строке **90** и на экран выводится значение **s**; в противном случае выводится значе-

ние t, управление передается оператору **end** в стоке **99** и выполнение программы завершается.

```
10 rem determine and print the sum of two integers
15 rem
20 rem input the two integers
30 input a
40 input b
45 rem
50 rem add integers and store result in c
60 let c = a + b
65 rem
70 rem print the result
80 print c
90 rem terminate program execution
99 end
```

Рис. 12.24. *Определение суммы двух целых*

```
10 rem determine and print the larger of two integers
20 input s
30 input t
32 rem
35 rem test if s >= t
40 if s >= goto 90
45 rem
50 rem t is greater than s, so print t
60 print t
70 goto 99
75 rem
80 rem s is greater than or equal to t, so print s
90 print s
99 end
```

Рис. 12.25. Нахождение большего из двух целых

Simple не поддерживает циклических конструкций (таких как **for**, **while** или **do/while** в языке C). Однако язык позволяет эмулировать любую из циклических конструкций C, используя операторы **if/goto** и **goto**. На рис. 12.26 показана программа, использующая цикл с контрольным значением для вычисления квадратов вводимых чисел. Каждое число вводится с клавиатуры и сохраняется в переменной **j**. Если вводится контрольное значение **-9999**, управление передается строке **99** и программа завершается. В противном случае переменной **k** присваивается значение переменной **j**, возведенное в квадрат, **k** выводится на экран и управление передается строке **20**, где вводится следующее число.

Используя программы на рис. 12.24, рис. 12.25 и рис. 12.26 в качестве образца, напишите на Simple программы, позволяющие выполнять следующие действия:

a) Ввести три числа, определить для них среднее значение и вывести результат на печать.

b) Используя цикл с контрольным значением, ввести десять чисел, вычислить их сумму и вывести полученное значение на печать.

```
10 rem calculate the squares of several integers
20 input j
23 rem
25 rem test for sentinel value
30 if j == -9999 goto 99
33 rem
35 rem calculate square of j and assign result to k
40 let k = j * j
50 print k
53 rem
55 rem loop to get next j
60 goto 20
99 end
```

Рис. 12.26. Вычисление квадрата вводимых пользователем чисел

c) Используя цикл с управлением по счетчику, ввести семь чисел, вычислить их сумму и вывести полученное значение на печать.

d) Ввести ряд чисел, определить среди них наибольшее и вывести его на печать. Первое введенное число должно показывать, сколько чисел должно быть обработано.

e) Ввести десять чисел и вывести на печать наименьшее из них.

f) Вычислить и вывести на печать сумму четных чисел от 2 до 30.

g) Вычислить и вывести на печать произведение нечетных чисел от 1 до 9.

12.27. *(Создание компилятора. Предварительные требования: выполнить упражнения 7.18, 7.19, 12.12, 12.13 и 12.26)* Теперь, когда описан язык Simple (упражнение 12.26), мы обсудим, как нам создать для него компилятор. Сначала рассмотрим процедуру, посредством которой программа Simple преобразуется в код SML и выполняется симулятором Симплетрона (см. рис. 12.27). Файл, содержащий программу на Simple, считывается компилятором и преобразуется в код SML. SML-код выводится в файл на диске, по одной инструкции SML на строку. Затем SML-файл загружается в симулятор Симплетрона, причем результат записывается в файл на диске и выводится на экран. Заметим, что программа Симплетрона, разработанная в упражнении 7.19, принимает данные с клавиатуры. Поэтому ее необходимо доработать для считывания из файла, чтобы она могла выполнять программу, созданную нашим компилятором.

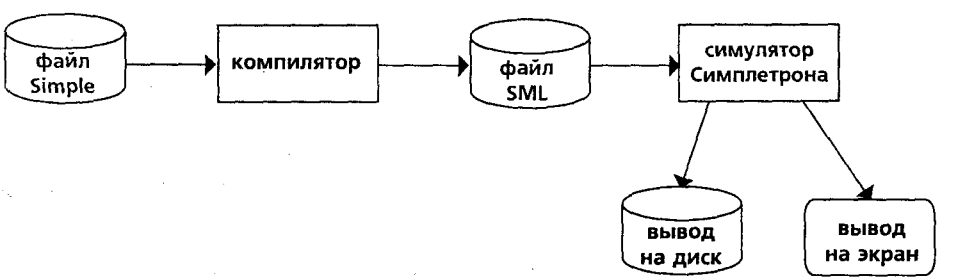


Рис. 12.27. Запись, компиляция и выполнение программы на языке Simple

Компилятор выполняет преобразование программы из Simple в SML за два *прохода*. Во время первого прохода создается *таблица символов*, в которой любые *номера строк*, *имена переменных* и *константы* программы Simple сохраняются со своим типом и соответствующим *положением* в конечном коде SML (таблица символов будет детально обсуждаться ниже). Кроме того, во время первого прохода для каждого оператора Simple вырабатываются соответствующие инструкции SML. Как будет видно из дальнейшего, если программа Simple содержит оператор, который передает управление строке, расположенной ниже, часть инструкций полученной SML-программы после первого прохода окажутся неполными. Во время второго прохода неполные инструкции находятся компилятором и дополняются, а конечная программа SML выводится в файл.

Первый проход

Компилятор начинает считывание операторов программы на Simple в память. Для обработки и последующей компиляции строки должны быть разделены на *лексемы* (т.е. «части» оператора; для этих целей может быть использована стандартная библиотечная функция **strtok**). При этом не следует забывать, что каждый оператор начинается с номера строки, предшествующего команде. По мере дробления оператора на лексемы, если лексема является номером строки, переменной или константой, она помещается в таблицу символов. Номер строки помещается в таблицу символов, только если это первая лексема оператора. **symbolTable** — это массив структур **tableEntry**, представляющих символы программы. Ограничения на число символов в программе отсутствуют. Следовательно, для отдельных программ **symbolTable** может оказаться очень большим. Определим на данный момент **symbolTable** как массив из 100 элементов. Вы можете увеличить или уменьшить этот размер, если программа работает.

Структура **tableEntry** определяется следующим образом:

```
struct tableEntry {
   int symbol;
   char type;  /* 'C', 'L', or 'V' */
   int location; /* 00 to 99 */
}
```

Каждая структура **tableEntry** содержит три элемента. Элемент **symbol** целого типа содержит ASCII-представление переменной (напомним, что имена переменных состоят из одного символа), номер строки или константу. Элемент **type** — это одна из следующих букв, обозначающих тип символа: **'C'** — для константы, **'L'** — для номера строки или **'V'** — для переменной. Элемент **location** содержит номер ячейки памяти Симплетрона (от **00** до **99**), на которую ссылается символ. Память Симплетрона — это массив из 100 целых, в котором хранятся инструкции SML и данные. Для номера строки ячейкой памяти является элемент в массиве памяти Симплетрона, в котором начинаются инструкции SML для оператора Simple. Для переменной или константы ячейкой памяти является элемент в массиве памяти Симплетрона, в котором хранится переменная или константа. Переменные и константы размещаются от

конца памяти Симплетрона к ее началу. Первая переменная или константа сохраняется в ячейке **99**, следующая **98** и так далее.

Таблица символов играет центральную роль в преобразовании написанных на Simple программ в код SML. В главе 7 мы узнали, что инструкция SML представляет собой четырехзначное целое число, состоящее из двух частей — *кода операции* и *операнда*. Код операции определяется командой Simple. Например, команда Simple **input** соответствует коду SML **10** (операция чтения), а команда Simple **print** соответствует коду SML **11** (операция записи). Операнд — это ячейка памяти, содержащая данные, над которыми выполняется операция (т.е. код операции **10** считывает значение, введенное с клавиатуры, и сохраняет его в заданной операндом ячейке памяти). Компилятор производит поиск в **symbolTable**, чтобы найти соответствующую символу ячейку памяти Симплетрона, которая затем используется для окончательного формирования инструкций SML.

Компиляция каждого оператора Simple зависит от содержащейся в нем команды. Например, после того, как номер строки для оператора **rem** вставляется в таблицу символов, оставшаяся часть оператора игнорируется компилятором, поскольку комментарий служит только для целей документирования. Операторы **input**, **print**, **goto** и **end** соответствуют инструкциям SML *read*, *write*, *branch* (переход к определенной ячейке памяти) и *halt* соответственно. Операторы, содержащие эти команды Simple, преобразуются непосредственно в инструкции SML (отметим, что оператор **goto** может содержать неразрешенную ссылку, если номер строки, на который он ссылается, находится дальше в файле программы Simple; так называемая ссылка вперед).

Когда оператор **goto** компилируется с неразрешенной ссылкой, инструкция SML должна быть *отмечена*, чтобы во время второго прохода компилятор дополнил инструкцию. Такие метки хранятся в массиве **flags** типа **int**, состоящем из 100 элементов, в котором каждому элементу при инициализации присвоено значение **-1**. Если ячейка памяти, на которую ссылается номер строки программы Simple, еще неизвестна (т.е. ее нет в таблице символов), номер строки сохраняется в массиве **flags** в элементе с тем же индексом, что и незаконченная инструкция. Операнд незаконченной инструкции временно устанавливается равным **00**. Например, инструкция безусловного перехода (выполняющая ссылку вперед) остается в виде **+4000** до второго прохода компилятора. Далее мы коротко остановимся на втором проходе компилятора.

Компиляция операторов **if/goto** и **let** сложнее по сравнению с другими операторами — это единственные операторы, транслируемые в более чем одну инструкцию SML. Для оператора **if/goto** компилятор генерирует код проверки условия и, в случае необходимости, перехода на другую строку. Результатом перехода может оказаться неразрешенная ссылка. Каждую из операций отношения и равенства можно смоделировать с помощью инструкций SML *перехода по нулю* и *перехода по минусу* (а возможно, комбинацией из этих двух переходов).

Для оператора **let** компилятор генерирует код вычисления арифметического выражения произвольной сложности, состоящего из целых переменных и/или констант. В выражении все операнды и операции должны быть отделены друг от друга пробелами. В упражнениях 12.12 и 12.13 были изложены алгоритмы преобразования инфиксных выражений в постфиксные и вычисления постфиксных выражений, применяемых в компиляторах для оценки арифметических выражений. Прежде чем приниматься за разработку своего компилятора, вы должны выполнить оба этих упражнения. Когда компилятор оценивает выражение, он сначала преобразует инфиксную нотацию в постфиксную, и после этого вычисляет постфиксное выражение.

А как компилятор производит машинный код для оценки выражения, содержащего переменные? Алгоритм постфиксной оценки содержит «крючок», который позволяет нашему компилятору создавать инструкции SML, а не выполнять реальные вычисления. Чтобы использовать вкомпиляторе этот «крючок», алгоритм постфиксной оценки необходимо изменить, чтобы он мог выполнять поиск в таблице символов каждого символа, который ему встречается (и, возможно, вставлять его в таблицу), определять соответствие символов ячейкам памяти и *помещать в стек эти ячейки памяти вместо символов*. В постфиксном выражении для выполнения операции из стека выталкиваются две ячейки памяти, и генерируется реализующий операцию машинный код с использованием этих ячеек в качестве операндов. Результат каждого подвыражения сохраняется во временной ячейке памяти и помещается обратно в стек, чтобы оценка постфиксного выражения могла продолжаться. Когда постфиксная оценка завершена, ячейка памяти, содержащая результат, оказывается единственной ячейкой, оставшейся в стеке. Она извлекается оттуда и создается инструкция SML, присваивающая результат переменной, стоящей в левой части оператора **let**.

Второй проход

Во время второго прохода компилятор выполняет две задачи: разрешает все неопределенные ссылки и выводит код SML в файл. Разрешение ссылок происходит следующим образом:

1) Производится поиск неразрешенных ссылок в массиве **flags** (т.е. поиск элементов, отличных от **-1**).

2) В массиве **symbolTable** ищется структура, содержащая символ, записанный в массиве **flags** (убедитесь, что это символ типа '**L**' для номера строки).

3) Из элемента структуры **location** в инструкцию с неразрешенной ссылкой вставляется ячейка памяти (напомним, что инструкция, содержащая неразрешенную ссылку, имеет операнд **00**).

4) Повторяются шаги 1, 2 и 3 до тех пор, пока не будет просмотрен весь массив **flags**.

После того как процесс разрешения завершен, весь массив, содержащий код SML, выводится в файл на диске по одной инструкции SML в строке. Теперь этот файл можно прочитать Симплетроном

для его исполнения (после того, как симулятор будет соответствующим образом усовершенствован и сможет использовать файл для ввода данных).

Законченный пример

Следующий пример иллюстрирует полное преобразование написанной на Simple программы в SML, так, как оно будет выполнено компилятором Simple. Рассмотрим программу на Simple, которая получает с клавиатуры целое число, и суммирует числа от 1 до этого числа. Программа и инструкции SML, созданные после первого прохода, показаны на рис. 12.28. Таблица символов, созданная после первого прохода, показана на рис. 12.29.

Программа на Simple	Ячейки памяти и инструкции SML	Описание
5 rem sum 1 to x	*нет*	**rem** игнорируется
10 input x	00 +1099	считывание **x** в ячейку **99**
15 rem check y == x	*нет*	**rem** игнорируется
20 if y == x goto 60	01 +2098	загрузка **y(98)** в аккумулятор
	02 +3199	вычитание **x(99)** из аккумулятора
	03 +4200	*переход по нулю к неразрешенной ячейке*
25 rem increment y	*нет*	**rem** игнорируется
30 let y = y + 1	04 +2098	загрузка **y** в аккумулятор
	05 +3097	прибавление **1(97)** к аккумулятору
	06 +2196	сохранение во временной ячейке **96**
	07 +2096	загрузка из временной ячейки **96**
	08 +2198	сохранение аккумулятора в **y**
35 rem add y to total	*нет*	**rem** игнорируется
40 let t = t + y	09 +2095	загрузка **t(95)** в аккумулятор
	10 +3098	прибавление **y** к аккумулятору
	11 +2194	сохранение во временной ячейке **94**
	12 +2094	загрузка из временной ячейки **94**
	13 +2195	сохранение аккумулятора в **t**
45 rem loop y	*нет*	**rem** игнорируется
50 goto 20	14 +4001	переход к ячейке **01**
55 rem output result	*нет*	**rem** игнорируется
60 print t	15 +1195	вывод **t** на экран
99 end	16 +4300	окончание выполнения программы

Рис. 12.28. Созданные после первого прохода компилятора инструкции SML

Символ	Тип	Ячейка памяти
5	L	00
10	L	00
'x'	V	99
15	L	01
20	L	01
'y'	V	98
25	L	04
30	L	04
1	C	97
35	L	09
40	L	09
't'	V	95
45	L	14
50	L	14
55	L	15
60	L	15
99	L	16

Рис. 12.29. Таблица символов для программы на рис. 12.28

Большинство операторов Simple преобразуются непосредственно в одну инструкцию SML. Исключение в этой программе составляют комментарии, оператор **if/goto** в строке **20** и операторы **let**. Комментарии не транслируются в машинные коды. Однако номер строки комментария помещается в таблицу символов на тот случай, если на этот номер будет ссылаться оператор **goto** или оператор **if/goto**. Строка **20** программы означает, что если условие **y==x** истинно, то управление передается строке **60**. Поскольку строка **60** появится в программе позже, во время первого прохода компилятора **60** не помещается в таблицу символов (номера строк помещаются в таблицу символов, только когда они встречаются в качестве первой лексемы в операторе). Следовательно, невозможно в этот момент определить операнд в инструкции SML *переход по нулю* в ячейке **03** массива инструкций SML. Компилятор помещает **60** в ячейку **03** массива **flags**, чтобы показать, что во время второго прохода компилятора необходимо завершить эту инструкцию.

Мы должны отслеживать положение следующей инструкции SML в массиве, поскольку нет однозначного соответствия между операторами Simple и инструкциями SML. Например, оператор **if/goto** в строке **20** компилируется в три инструкции SML. Каждый раз, когда создается инструкция SML, мы должны увеличивать *счетчик*

инструкций, перемещаясь в следующую ячейку массива SML. Отметим, что размер памяти Симплетрона может представлять проблему для программ Simple, содержащих большое количество операторов, переменных и констант. Вполне возможно, что компилятору не хватит памяти. Чтобы иметь возможность обнаружить такую ситуацию, ваша программа должна содержать *счетчик данных*, чтобы отслеживать ячейку памяти в массиве SML, в которой будет сохраняться следующая константа или переменная. Если значение счетчика инструкций больше, чем значение счетчика данных, массив SML заполнен. В этом случае процесс компиляции необходимо прервать и компилятор должен выдать сообщение об ошибке, что в процессе компиляции обнаружилась нехватка памяти.

Пошаговый просмотр процесса компиляции

Давайте теперь пройдем процесс компиляции программы Simple, представленной на рис. 12.28, по шагам. Компилятор считывает в память первую строку программы:

```
5 rem sum 1 to x
```

С помощью **strtok** выделяется первая лексема оператора — номер строки (см. в главе 8 описание функций С, работающих со строками). Лексема, возвращенная **strtok**, преобразуется с помощью **atoi** в целое, чтобы символ **5** можно было искать в таблице символов. Если символ не обнаружен, он вставляется в таблицу. Так как мы находимся в самом начале программы, в таблице пока нет ни одного символа. Поэтому **5** вставляется в таблицу символов, как принадлежащее к типу **L** (номер строки), и ему присваивается первая ячейка памяти в массиве SML (**00**). Хотя эта строка является комментарием, для нее все равно выделяется место в таблице символов (на тот случай, если на нее будут ссылаться операторы **goto** или **if/goto**). Для оператора **rem** не создается никаких инструкций SML, поэтому значение счетчика инструкций не изменяется.

Далее разбивается на лексемы оператор

```
10 input x
```

Номер строки **10** помещается в таблицу символов как имеющий тип **L** и ему присваивается первая ячейке массива SML (**00**, поскольку программа начиналась с комментария и счетчик инструкций по-прежнему равен **00**). Команда **input** означает, что следующая лексема — переменная (только переменная может присутствовать в операторе **input**). Поскольку **input** непосредственно соответствует коду операции SML, компилятор просто определяет положение **x** в массиве SML. Символ **x** не обнаружен в таблице символов. Поэтому он включается в таблицу символов как ASCII-представление буквы **x**, получает тип **V**, и ему присваивается ячейка **99** в массиве SML (данные начинают размещаться с ячейки **99**, и ячейки памяти для них выделяются в обратном направлении). Теперь для этого оператора можно генерировать код SML. Код **10** (код операции считывания в SML) умножается на 100, и к получившемуся значению прибавляется ячейка таблицы символов, в которой расположен **x**, образуя полную инструкцию. Отметим, что инструкция сохраняется в массиве SML в ячейке **00**. Счетчик инструкций увеличивается на 1, поскольку была создана одна инструкция SML.

Оператор

```
15 rem check y == x
```

разбивается на лексемы. В таблице символов ищется строка с номером **15** (которой там нет). Номер строки помещается в таблицу, как имеющий тип **L**, и ему присваивается следующая ячейка массива, **01** (напомним, что оператор **rem** не производит кода, поэтому счетчик инструкций сохраняет свое значение).

Оператор

```
20 if y == x goto 60
```

разбивается на лексемы. Номер строки **20** вставляется в таблицу символов и получает тип **L** и следующую ячейку массива SML — **01**. Команда **if** означает, что будет производиться проверка на выполнение условия. Переменная **y** не обнаружена в таблице символов, поэтому она помещается туда, получает тип **V** и ячейку **98**. Затем генерируются инструкции SML для оценки условия. Поскольку в SML отсутствует прямой эквивалент для оператора **if/goto**, его приходится эмулировать, выполняя вычисления с **x** и **y** и ветвление в зависимости от результата. Если **x** равен **y**, результат вычитания **x** из **y** равен нулю, поэтому для эмуляции оператора **if/goto** можно использовать инструкцию перехода по нулю. Первый шаг требует, чтобы значение **y** было загружено (из SML ячейки с номером **98**) в аккумулятор. Для этого создается инструкция **01 +2098**. Следующим шагом из аккумулятора вычитается **x**. Для этого создается инструкция **02 +3199**. Значение в аккумуляторе может оказаться как нулевым, так и положительным или отрицательным. Так как рассматривается операция ==, нас интересует *переход по нулю*. Сначала в таблице символов ищется ячейка перехода (в данном случае **60**), которая не находится. Поэтому **60** помещается в ячейку **03** массива **flags**, и создается инструкция **03 +4200** (мы не можем указать ячейку перехода, поскольку еще не присвоили строке **60** ячейку в массиве SML). Счетчик инструкций увеличивает свое значение до **04**.

Компилятор переходит к оператору

```
25 rem increment y
```

Строка номер **25** вставляется в таблицу символов, как имеющая тип **L**, и получает SML-ячейку **04**. Счетчик инструкций не увеличивается.

Когда происходит разбиение на лексемы оператора

```
30 let y = y + 1
```

номер строки **30** вставляется в таблицу символов как имеющий тип **L** и ему присваивается ячейка SML **04**. Команда **let** означает, что данная строка является оператором присваивания. Сначала все символы строки помещаются в таблицу символов (если они еще там не находятся). Число **1** добавляется в таблицу как имеющее тип **C** и ему выделяется ячейка **97**. Далее та часть, что расположена справа от оператора присваивания, переводится из инфиксной формы в постфиксную. После этого вычисляется постфиксное выражение (**y 1 +**). В таблице символов ищется символ **y**, и соответст-

вующая ему ячейка памяти помещается в стек. Символ **1** также ищется в таблице символов и соответствующая ячейка также помещается в стек. Встретив знак операции +, постфиксный вычислитель извлекает из стека правый и левый операнды, после чего создаются инструкции SML

```
04 +2098 (load y)
05 +3097 (add 1)
```

Результат выражения сохраняется во временной ячейке памяти **(96)** инструкцией

```
06 +2196 (store temporary)
```

и адрес временной ячейки помещается в стек. Теперь, когда выражение оценено, результат необходимо сохранить в **y** (т.е. переменной, стоящей слева от =). Поэтому значение из временной ячейки загружается в аккумулятор и аккумулятор сохраняется в **y** при помощи инструкций

```
07 +2096 (load temporary)
08 +2198 (store y)
```

Читатель, конечно, тут же заметит, что созданные инструкции SML избыточны. В дальнейшем мы еще вернемся к этой проблеме.

Когда оператор

```
35 rem add y to total
```

разбивается на лексемы, номер строки **35** вставляется в таблицу символов, как имеющий тип **L** и ему выделяется ячейка **09**.

Оператор

```
40 let t = t + y
```

аналогичен оператору в строке **30**. Переменная **t** помещается в таблицу символов, как имеющая тип **V**, и ей присваивается ячейка SML **95**. Инструкции следуют той же самой логике и формату, что и для строки **30**, и при этом создаются инструкции **09 +2095**, **10 +3098**, **11 +2194** и **13 +2195**. Отметим, что результат выполнения операции **t + y** присваивается временной ячейке **94**, перед тем как будет присвоен **t (95)**. Читатель, конечно же, опять заметил, что инструкции в ячейках **11** и **12** избыточны. Как уже говорилось, мы вернемся к этой теме позже.

Оператор

```
45 rem loop y
```

является комментарием, поэтому строка **45** добавляется в таблицу символов как имеющая тип **L**, и ей присваивается ячейка SML **14**.

Оператор

```
50 goto 20
```

передает управление строке **20**. Номер строки **50** помещается в таблицу символов с типом **L** и ему присваивается ячейка **14**. Эквивалентом **goto** в SML является *безусловный переход* **(40)** — инструкция, передающая управление заданной ячейке SML. Компилятор просматривает таблицу символов в поисках строки **20** и обнаруживает, что она соответствует ячейке SML **01**. Код операции **(40)** ум-

ножается на 100 и к нему прибавляется ячейка **01**; в результате получается инструкция **14 +4001**.

Оператор

```
55 rem output result
```

является комментарием, поэтому строка **55** добавляется в таблицу символов как имеющая тип **L**, и ей присваивается ячейка SML **15**.

Оператор

```
60 print t
```

является оператором вывода. Номер строки **60** вставляется в таблицу символов как имеющий тип **L** и соответствующий ячейке **15**. Аналогом **print** в SML служит код операции **11** (*запись*). Ячейка, содержащая **t**, находится в таблице символов и складывается с кодом операции, умноженным на 100.

Оператор

```
99 end
```

является последней строкой программы. Номер строки **99** вставляется в таблицу символов как имеющий тип **L**, и ему присваивается ячейка **16**. Команда **end** генерирует инструкцию SML **+4300** (**43** в SML означает *стоп*), которая записывается как последняя инструкция в массиве памяти SML.

На этом первый проход компилятора завершается. Теперь рассмотрим второй проход. Он начинается с поиска в массиве **flags** значений, отличных от **-1**. Ячейка **03** содержит **60**, таким образом, компилятор узнает, что инструкция **03** неполна. Компилятор завершает инструкцию, находя в таблице символов **60**, определяя соответствующую ячейку и прибавляя ее адрес к неполной инструкции. В данном случае оказывается, что **60** соответствует ячейке SML **15**, поэтому **03 +4200** заменяется законченной инструкцией **03 +4215**. Теперь компиляция написанной на Simple программы успешно завершена.

Чтобы создать компилятор, вам необходимо решить следующие задачи:

a) Усовершенствовать программу Симплетрона, которую вы написали в упражнении 7.19, чтобы она могла считывать данные с заданного пользователем файла (см. главу 11). Кроме того, симулятор должен выводить результаты в файл на диске в том же формате, как они выводятся на экран.

b) Модифицировать данный в упражнении 12.12 алгоритм преобразования инфиксного выражения в постфиксное, чтобы он мог работать с многозначными целыми операндами и операндами в виде однобуквенной переменной. Подсказка: для выделения каждой константы и переменной в выражении можно воспользоваться стандартной библиотечной функцией **strtok**, а преобразование констант из строк в числа выполнять с помощью стандартной функции **atoi**. (Замечание: представление данных в постфиксном выражении должно быть изменено, чтобы можно было работать с именами переменных и целыми константами.)

c) Модифицировать алгоритм вычисления постфиксного выражения, чтобы он мог работать с операндами в виде многозначных чисел и имен переменных. Кроме того, алгоритм должен иметь обсуждавшийся выше «крючок», чтобы вместо непосредственного вычисления выражения создавались инструкции SML. Подсказка: чтобы выделить в выражении отдельные константы и переменные, можно воспользоваться функцией стандартной библиотеки **strtok**, после чего константы можно преобразовать из строк в числа с помощью функции стандартной библиотеки **atoi** (Замечание: представление данных в постфиксном выражении должно быть изменено, чтобы была возможность работать с именами переменных и целыми константами.).

d) Создание компилятора. Объедините части (b) и (c) для оценки выражений в операторах **let**. Ваша программа должна содержать функцию, выполняющую первый проход компилятора, и функцию, выполняющую второй проход. Обе функции для выполнения своих задач могут вызывать другие функции.

12.28. *(Оптимизация компилятора Simple)* После того, как программа была компилирована и преобразована в SML, мы получили набор инструкций. Некоторые комбинации инструкций часто повторяются, обычно в триплетах, называемых *порождениями*. Порождения, как правило, состоят из трех инструкций, таких как *загрузить*, *прибавить* и *сохранить*. Например, на рис. 12.30 представлены пять инструкций SML, созданных при компиляции программы из рис. 12.28. Первые три инструкции составляют порождение, которое прибавляет **1** к **y**. Заметим, что инструкции **06** и **07** сохраняют значение, находящееся в аккумуляторе, во временной ячейке **96**, после чего вновь загружают это значение в аккумулятор, чтобы инструкция **08** могла сохранить значение в ячейке **98**. Часто за порождением следует инструкция загрузки той же самой ячейки, в которую только что производилась запись. Такой код можно *оптимизировать* путем удаления инструкции сохранения и следующей непосредственно за ней инструкции загрузки, оперирующей с той же самой ячейкой памяти. Такая оптимизация позволит Симплетрону быстрее выполнять программу, так как в ней будет меньше инструкций. На рис. 12.31 показана оптимизированная программа SML, полученная из представленной ранее на рис. 12.28. Отметим, что в оптимизированном коде на четыре инструкции меньше, так что экономия памяти составляет 25%.

```
04 +2098  (load)
05 +3097  (add)
06 +2196  (store)
07 +2096  (load)
08 +2198  (store)
```

Рис. 12.30. Не оптимизированный код программы, представленной на рис. 12.28

Усовершенствуйте компилятор, введя в него оптимизацию создаваемого кода Машинного языка Симплетрона. Вручную сравните оптимизированный и не оптимизированный код и посчитайте, на сколько процентов оптимизированный код короче.

Программа на Simple	Ячейки памяти и инструкции SML	Описание
5 rem sum 1 to x	*нет*	**rem** игнорируется
10 input x	00 +1099	считывание **x** в ячейку **99**
15 rem check y == x	*нет*	**rem** игнорируется
20 if y == x goto 60	01 +2098	загрузка **y(98)** в аккумулятор
	02 +3199	вычитание **x(99)** из аккумулятора
	03 +4211	переход по нулю к ячейке **11**
25 rem increment y	*нет*	**rem** игнорируется
30 let y = y + 1	04 +2098	загрузка **y** в аккумулятор
	05 +3097	прибавление **1(97)** к аккумулятору
	06 +2198	сохранение аккумулятора в **y(98)**
35 rem add y to total	*нет*	**rem** игнорируется
40 let t = t + y	07 +2096	загрузка **t** из ячейки (**96**)
	08 +3098	прибавление **y(98)** к аккумулятору
	09 +2196	сохранение аккумулятора в **t(96)**
45 rem loop y	*нет*	**rem** игнорируется
50 goto 20	10 +4001	переход к ячейке **01**
55 rem output result	*нет*	**rem** игнорируется
60 print t	11 +1196	вывод **t(96)** на экран
99 end	12 +4300	окончание выполнения программы

Рис. 12.31. Оптимизированный код для программы на рис. 12.28

12.29. (*Усовершенствование компилятора языка Simple*) Выполните следующие модификации компилятора Simple. Некоторые из них могут также потребовать усовершенствования программы-симулятора Симплетрона, написанной в упражнении 7.19.

a) Сделать возможным использование операции взятия по модулю (%) в операторе **let**. Необходимо внести изменения и в Машинный язык Симплетрона, дополнив его инструкцией для взятия по модулю.

b) Сделать возможным использование операции возведения в степень (^) в операторе **let**. Необходимо внести изменения и в Машинный язык Симплетрона, дополнив его инструкцией для возведения в степень.

c) Сделать возможным распознавание компилятором букв как верхнего, так и нижнего регистра (т.е. написание '**A**' и '**a**' должно приводить к одинаковому результату). Никаких изменений в симуляторе Симплетрона не требуется.

d) Сделать возможным считывание оператором **input** сразу нескольких переменных, например: **input x, y**. Никаких изменений в симуляторе Симплетрона не требуется.

e) Сделать возможным вывод сразу нескольких переменных с помощью одного оператора **print**, например: **print a, b, c**. Никаких изменений в симуляторе Симплетрона не требуется.

f) Ввести проверку синтаксиса, чтобы компилятор выводил сообщение об ошибке всякий раз, когда в программе Simple встречается синтаксическая ошибка. Никаких изменений в симуляторе Симплетрона не требуется.

g) Сделать возможным использование массивов чисел. Никаких изменений в симуляторе Симплетрона не требуется.

h) Сделать возможным использование подпрограмм, управляемых командами Simple **gosub** и **return**. Команда **gosub** передает управление подпрограмме, а команда **return** возвращает его обратно оператору, следующему за **gosub**. Это будет что-то похожее на вызов функции в C. Допускается вызов одной и той же подпрограммы несколькими **gosub**, расположенными в разных местах программы. Никаких изменений в симуляторе Симплетрона не требуется.

i) Сделать возможным использование структур повторения вида

```
for x = 2 to 10 step 2
   операторы Simple
next
```

Этот оператор **for** организует цикл от **2** до **10** с шагом **2**. Оператор **next** отмечает конец тела цикла, начинающегося строкой **for**. Никаких изменений в симуляторе Симплетрона не требуется.

j) Сделать возможным использование структур повторения вида

```
for x = 2 to 10
   операторы Simple
next
```

Такой оператор **for** организует цикл от **2** до **10** с шагом **1**. Никаких изменений в симуляторе Симплетрона не требуется.

k) Сделать возможной обработку вывода и ввода строк. При этом потребуется соответствующим образом изменить симулятор Симплетрона, чтобы он мог обрабатывать и хранить данные строкового типа. Подсказка: каждое слово Симплетрона можно разделить на две части, каждая из которых содержит двузначное целое. Каждое двузначное целое представляет десятичный эквивалент символа ASCII. Добавьте инструкцию машинного языка, которая будет печатать строку, начинающуюся в некоторой ячейке памяти Симплетрона. Первая половина слова в этой ячейке является числом символов строки (т.е. ее длиной). Каждое последующее полуслово содержит один символ ASCII, представленный двумя десятичными цифрами. Инструкция машинного языка проверяет длину и печатает строку, переводя каждое двузначное число в соответствующий символ.

l) Сделать возможной обработку не только целых, но и чисел с плавающей точкой. Необходимо внести соответствующие изменения и в симулятор Симплетрона, чтобы он тоже мог работать с числами с плавающей точкой.

12.30. (*Интерпретатор Simple*) Интерпретатором называется программа, которая считывает операторы языка высокого уровня, определяет операции, которые они должны выполнить, и тут же эти операции выполняет. Программа не преобразуется предварительно в машинный код. Интерпретаторы работают медленно, поскольку каждый исполняемый оператор программы сначала необходимо дешифровать. Если оператор помещен в цикл, дешифровка оператора будет производиться при каждом его вызове в теле цикла. Ранние версии языка программирования BASIC были реализованы в виде интерпретаторов.

Напишите интерпретатор для языка Simple, обсуждавшегося в упражнении 12.26. Для оценки выражений в операторе **let** программа должна использовать инфиксно-постфиксный конвертор, разработанный в упражнении 12.12, и вычислитель постфиксных выражений, разработанный в упражнении 12.13. Ограничения, наложенные на язык Simple в упражнении 12.26, должны относиться и к этой программе. Протестируйте интерпретатор, используя программы Simple из упражнения 12.26. Сравните результаты выполнения этих программ в интерпретаторе с результатами компиляции программ Simple и запуска их в симуляторе Симплетрона, разработанном в упражнении 7.19.

глава



Препроцессор



Цели

- Научиться применению директивы **#include** при создании больших программ.
- Научиться применению директивы **#define** для определения макросов и макросов с аргументами.
- Познакомиться с условной компиляцией.
- Научиться выводить сообщения об ошибках с применением условной компиляции.
- Научиться использованию макросов для проверки корректности значений.

Содержание

13.1. Введение
13.2. Директива препроцессора #include
13.3. Директива препроцессора #define: символические константы
13.4. Директива препроцессора #define: макросы
13.5. Условная компиляция
13.6. Директивы препроцессора #error и #pragma
13.7. Операции # и ##
13.8. Нумерация строк
13.9. Предопределенные символические константы
13.10. Макрос подтверждения

Резюме • Распространенные ошибки программирования • Хороший стиль программирования • Советы по повышению эффективности • Упражнения для самоконтроля • Ответы на упражнения для самоконтроля • Упражнения

13.1. Введение

Эта глава — введение в *препроцессор* C, выполняющий, как это следует из его названия, обработку программы до ее компиляции. Вот некоторые из возможных действий препроцессора: включение других файлов в файл, который будет компилироваться, определение *символических констант* и *макросов*, *условная компиляция* кода программы и *условное выполнение директив препроцессора*. Все директивы препроцессора начинаются с #, и только пробельные символы могут стоять в строке перед этими директивами.

13.2. Директива препроцессора #include

Директива препроцессора ***#include*** использовалась на протяжении всей книги. Директива **#include** создает копию указанного файла, которая включается в программу вместо директивы. Существует две формы использования директивы **#include**:

```
#include <filename>
#include "filename"
```

Разница между ними заключается в том, где препроцессор будет искать файлы, которые необходимо включить. Если имя заключено в кавычки, препроцессор ищет его в том же каталоге, что и компилируемый файл. Такую запись обычно используют для включения определенных пользователем заголовочных файлов. Если же имя файла заключено в угловые скобки (< и >) — используемые для *файлов стандартной библиотеки*, — то поиск будет вестись в зависимости от конкретной реализации компилятора, обычно в предопределенных каталогах.

Директива **#include** чаще всего используется для включения заголовочных файлов стандартной библиотеки, таких как **stdio.h** и **stdlib.h** (см. рис. 5.6). Директива **#include** также используется с программами, состоящими из нескольких файлов, которые необходимо компилировать вместе. Часто с помощью этой директивы включается заголовочный файл, содержащий общие для отдельных файлов программы объявления. Примерами таких объявлений являются объявления структур и объединений, перечислимых констант и прототипов функций.

В системе UNIX программные файлы компилируются посредством *команды cc*. Например, компилирование и компоновка **main.c** и **square.c** производится при помощи команды

```
cc main.c square.c
```

введенной в командной строке UNIX. При этом создается выполняемый файл **a.out**. Чтобы получить более подробную информацию о процессе компиляции, компоновки и выполнения программ, воспользуйтесь руководством к вашему компилятору.

13.3. Директива препроцессора #define: символические константы

Директива ***#define*** создает *символические константы* — константы, представленные символами, и *макросы* — операции, определенные как символы. Формат директивы **#define** выглядит следующим образом.

```
#define идентификатор заменяющий_текст
```

Когда такая строка появляется в файле, вместо всех последующих появлений *идентификатора* будет автоматически подставлен *заменяющий_текст*, перед тем как программа будет компилироваться. Например,

```
#define PI 3.14159
```

заменяет все последующие появления символической константы **PI** на численную константу **3.14159**. Символические константы позволяют программисту создавать имя для константы и использовать это имя в любом месте программы. Если необходимо изменить значение константы во всей программе, то достаточно сделать это в одном месте, в директиве **#define**, и после повторной компиляции программы все вхождения константы в программе будут автоматически изменены. Примечание: *при замене вместо именованной константы будет подставлено все, что стоит справа от нее в определении* ***#define***. Например, строка **#define PI = 3.14159** приведет к тому, что препроцессор заменит по всему тексту программы **PI** на **= 3.14159**. Подобные ошибки часто

приводят к многочисленным, трудно выявляемым логическим и синтаксическим ошибкам. Переопределение символической константы, при котором ей присваивается новое значение, также является ошибкой.

Хороший стиль программирования 13.1

Использование осмысленных имен для символических констант делает программу самодокументированной и облегчает ее восприятие.

13.4. Директива препроцессора #define: макросы

Макрос — операция, определяемая при помощи директивы препроцессора **#define**. Как и в случае символических констант, перед компиляцией программы вместо *идентификатора макроса* в программу подставляется *заменяющий_текст*. Допускается определение макроса с *аргументами* или без них. Макрос без аргументов обрабатывается так же, как и символическая константа. В макросе с аргументами последние подставляются в заменяющий текст, после чего макрос *расширяется*, то есть в программу подставляется *заменяющий_текст* вместо идентификатора и списка аргументов.

Рассмотрим следующее макроопределение с одним аргументом для нахождения площади круга:

```
#define CIRCLE_AREA(x) (PI * (x) * (x) )
```

Всякий раз, когда в файле появляется **CIRCLE_AREA(x)**, значение **x** подставляется вместо **x** в заменяющем тексте, символическая константа **PI** заменяется ее значением (определенным ранее), и макрос расширяется в программе. Например, оператор

```
area = CIRCLE_AREA(4);
```

расширяется в

```
area = ( 3.14159 * (4) * (4) );
```

Так как выражение состоит только из констант, значение выражения оценивается в процессе компиляции и присваивается переменной **area**. Скобки, в которые заключены **x** в заменяющем тексте, обеспечивают правильный порядок вычисления, когда в качестве аргумента макроса используется выражение. Например, оператор

```
area = CIRCLE_AREA(c + 2);
```

расширяется в выражение

```
area = (3.14159 * (c + 2) * (c + 2) );
```

которое оценивается правильно, поскольку скобки обеспечивают правильный порядок выполнения операций. Если бы скобок не было, макрос расширился бы в выражение

```
area = 3.14159 * c + 2 * c + 2;
```

которое оценивалось бы неправильно как

```
area = (3.14159 * c) + (2 * c) + 2;
```

согласно правилам старшинства операций.

Распространенная ошибка программирования 13.1

Забывают заключить аргументы в скобки в заменяющем тексте макроса.

Вместо макроса **CIRCLE_AREA** можно было использовать функцию. Функция **circleArea**

```
double circleArea(double x)
{
   return 3.14159 * x * x;
}
```

выполняет те же самые вычисления, что и макрос **CIRCLE_AREA**, но использование функции **circleArea**, по сравнению с макросом, приводит к определенным накладным расходам. Преимущества использования макроса **CIRCLE_AREA** состоят в том, что макрос вставляет код непосредственно в программу, помогая, таким образом, избежать накладных расходов, связанных с вызовом функции, и при всем при том программа остается удобочитаемой, поскольку вычисление **CIRCLE_AREA** определено отдельно и названо осмысленно. Недостатком является то, что аргумент этого макроса оценивается дважды.

Совет по повышению эффективности 13.1

Иногда вместо вызова функции можно использовать макрос, который встраивает код в программу до времени исполнения. Это исключает накладные расходы, связанные с вызовом функции.

Следующее определение макроса содержит два аргумента и предназначено для вычисления площади прямоугольника:

```
#define RECTANGLE_AREA(x, y) ( (x) * (y) )
```

Всякий раз, когда в тексте программы встречается **RECTANGLE_AREA (x, y)**, значения **x** и **y** подставляются в заменяющий текст макроса, и макрос расширяется в том месте, где стояло его имя. Например, оператор

```
rectArea = RECTANGLE_AREA(a + 4, b + 7);
```

расширяется в

```
rectArea = ( (a + 4) * (b + 7) );
```

Значение выражения оценивается и присваивается переменной **rectArea**.

Заменяющий текст для макроса или символической константы — это обычно любой текст в строке после идентификатора в директиве **#define**. Если заменяющий текст для макроса или именованной константы не умещается на одной строке, в конце строки следует поместить обратную косую черту (\), которая показывает, что текст продолжается на следующей строке.

Символические константы и макросы можно отменить, используя *директиву препроцессора* ***#undef***. Директива **#undef** отменяет определение символической константы или имени макроса. *Область действия* символической константы или макроса простирается от места их определения до места отмены определения с помощью **#undef**, или до конца файла. После отмены определения допускается его повторное определение с помощью **#define**.

Функции стандартной библиотеки иногда определяются как макросы на основе других библиотечных функций. В заголовочном файле **stdio.h** часто определяют макрос

```
#define getchar( ) getc(stdin)
```

Макроопределение **getchar** использует функцию **getc**, чтобы получить один символ из стандартного потока ввода. Функция **putchar** заголовочного файла **stdio.h** и оперирующие с символами функции заголовочного файла **cty-**

pe.h часто также реализуются в виде макросов. Заметим, что выражения с побочными эффектами (то есть с изменяемыми значениями переменных) не должны помещаться в макросы, потому что аргументы макроса могут оцениваться более одного раза.

13.5. Условная компиляция

Условная компиляция позволяет программисту управлять выполнением директив препроцессора и компиляцией программного кода. Каждая из условных директив препроцессора оценивает значение целочисленного выражения. В директивах препроцессора невозможна оценка выражений приведения типа, выражений **sizeof** и перечислимых констант.

Условные конструкции препроцессора во многом похожи на условный оператор **if**. Рассмотрим следующий код препроцессора:

```
#if !defined(NULL)
   #define NULL 0
#endif
```

Эти директивы устанавливают, определен ли **NULL**. Выражение **defined(NULL)** имеет значение **1**, если **NULL** определен, и **0** в противном случае. Если результат **0**, **!defined(NULL)** имеет значение **1** и выполняется определение **NULL**. В противном случае директива **#define** пропускается. Каждая конструкция **#if** заканчивается **#endif**. Директивы ***#ifdef*** и ***#ifndef*** — это сокращения для **#if defined(*имя*)** и **#if !defined(*имя*)**. Условные конструкции препроцессора, проверяющие несколько вариантов, реализуются с помощью директив **#elif** (эквивалента **else if** структуры **if**) и **#else** (эквивалента **else** структуры **if**).

При разработке программ программисты, чтобы предотвратить компиляцию больших кусков программы, часто превращают их в комментарии. Если сам код программы содержит комментарии, то **/*** и ***/** для этого не подходят. В этом случае можно использовать следующую конструкцию препроцессора:

```
#if 0
   код, компиляцию которого надо предотвратить
#endif
```

Чтобы снова разрешить компиляцию кода, следует заменить в представленной выше конструкции **0** на **1**.

Условная компиляция часто служит средством, помогающим отладить программу. Многие реализации C снабжены *отладчиками*. Однако отладчики, как правило, слишком сложны для начинающих программистов, и поэтому новички редко ими пользуются. Вместо этого используют операторы **printf**, чтобы вывести на печать значения переменных и убедиться в правильности хода выполнения программы. Если такие операторы **printf** заключить в директивы условной компиляции препроцессора, то их компиляция будет выполняться только в период отладки. Например,

```
#ifdef DEBUG
   printf("Variable x = %d\n", x);
#endif
```

приведет к тому, что оператор **printf** будет компилироваться в программе, если перед директивой **#ifdef DEBUG** была определена (**#define DEBUG**) символическая константа **DEBUG**. Когда отладка завершена, директива **#define** удаляет-

ся из исходного файла и операторы **printf**, вставленные для целей отладки, будут при компиляции игнорироваться. В больших программах желательно определить несколько различных символических констант, чтобы иметь возможность контролировать условия компиляции отдельных частей исходного файла.

Распространенная ошибка программирования 13.2

Вставка условно компилируемых операторов **printf** для целей отладки в места, где С подразумевает только один оператор. В этом случае условно компилируемый фрагмент должен быть включен в составной оператор. Тогда, если программа компилируется с отладочными операторами, поток управления не изменяется.

13.6. Директивы препроцессора #error и #pragma

Директива ***#error***

```
#error лексемы
```

выводит на печать зависящее от реализации сообщение, включая *лексемы*, определенные в директиве. Лексемы — это последовательности символов, разделенные пробелами. Скажем, директива

```
#error 1 - Out of range error
```

содержит шесть лексем. Например, в Borland C++ для PC, когда выполняется директива **#error**, лексемы в директиве отображаются как сообщение об ошибке; затем препроцессор останавливается и программа не компилируется.

Директива ***#pragma***

```
#pragma лексемы
```

указывает действие, зависящее от реализации. Указание, не распознанное данным компилятором, игнорируется. Например, в Borland C++ распознается несколько указаний, которые позволяют программисту полностью использовать преимущества, имеющиеся в Borland C++. Более полную информацию относительно использования **#error** и **#pragma** вы можете узнать из документации к вашему компилятору С.

13.7. Операции # и

Операции препроцессора # и ## имеются только в ANSI C. Операция # выполняет преобразование текстовой лексемы в строку, заключенную в кавычки. Рассмотрим следующее макроопределение:

```
#define HELLO(x) printf("Hello, " #x "\n");
```

Когда в файле программы появляется выражение **HELLO(John)**, оно расширяется в

```
printf("Hello, " "John" "\n");
```

Строка **"John"** подставляется в заменяющий текст вместо **#x**. Препроцессор производит конкатенацию строк, разделенных пробельными символами, поэтому предыдущий оператор эквивалентен

```
printf("Hello, John\n");
```

Заметим, что операция # должна использоваться в макросах с аргументами, поскольку операнд после # ссылается на аргумент макроса.

Операция ## выполняет конкатенацию двух лексем. Рассмотрим следующее макроопределение:

```
#define TOKENCONCAT(x, y) x ## y
```

Когда **TOKENCONCAT** появляется в тексте программы, происходит конкатенация его аргументов, после чего они заменяют макроопределение. Например, **TOKENCONCAT(O,K)** заменяется в программе на **OK**. Операция ## должна иметь два операнда.

13.8. Нумерация строк

Директива препроцессора ***#line*** используется для последовательной нумерации строк исходного кода программы, начиная с числа, заданного константой. Директива

```
#line 100
```

начинает нумерацию строк исходного кода программы со следующей за директивой строки, которой присваивается номер **100**. В директиву **#line** можно включить имя файла. Директива

```
#line 100 "file1.c"
```

означает, что строки, начиная со следующей за директивой **#line**, будут пронумерованы, нумерация начнется со **100** и что имя файла для любых сообщений компилятора "**file1.c**". Директива обычно используется для того, чтобы сделать сообщения компилятора о синтаксических ошибках и предупреждениях более содержательными. В исходном файле номера строк не появятся.

13.9. Предопределенные символические константы

Существует пять *предопределенных символических констант* (рис. 13.1.) Идентификаторы каждой из предопределенных констант начинаются и заканчиваются *двумя* символами подчеркивания. Эти идентификаторы, так же как и идентификатор **defined** (упоминавшийся в разделе 13.5), не могут входить в директивы **#define** или **#undef**.

13.10. Макрос подтверждения

Макрос ***assert***, определенный в заголовочном файле **assert.h**, проверяет значение выражения. Если значение выражения равно **0** (false), то **assert** распечатывает сообщение об ошибке и вызывает функцию ***abort*** (из библиотеки общего назначения **stdlib.h**), завершающую работу программы. Это полезный инструмент отладки, позволяющий проверить, имеет ли переменная верное значение. Допустим, переменная **x** в программе никогда не должна превышать значение **10**. Для проверки значения **x** и выдачи сообщения об ошибке в случае, если **x** принимает недопустимые значения, можно воспользоваться макросом подтверждения. Соответствующий оператор будет выглядеть следующим образом:

```
assert (x <= 10);
```

Если при выполнении этого оператора **x** больше **10**, выдается сообщение, содержащее номер строки и имя файла, а выполнение программы при этом прерывается. После этого программист может ограничить поиски ошибки данной частью кода. Если определена символическая константа **NDEBUG**, последующие операторы подтверждения будут игнорироваться. Таким образом, когда операторы подтверждения уже не нужны, вместо того, чтобы уничтожать их вручную, достаточно в программный файл вставить строку

```
#define NDEBUG
```

Символическая константа	Объяснение
__LINE__	*Номер текущей обрабатываемой строки исходного кода* программы (целая константа).
__FILE__	Имя компилируемого исходного файла (символьная строка).
__DATE__	Дата начала компиляции текущего исходного файла (строка вида **"Mmm dd yyyy"**, например **"Jan 19 1991"**).
__TIME__	Время начала компиляции текущего исходного файла (символьная строка вида **"hh:mm:ss"**).
__STDC__	*Целая константа 1. Показывает, что данная реализация совместима* со стандартом ANSI.

Рис. 13.1. Предопределенные символические константы

Резюме

- Все директивы препроцессора начинаются с #.
- Перед директивой препроцессора в строке могут стоять только пробельные символы.
- Директива **#include** включает в текст копию указанного файла. Если имя файла заключено в кавычки, препроцессор ищет его в том же каталоге, что и файл, который компилируется. Если же имя файла заключено в угловые скобки (< и >), место, где будет выполняться поиск, зависит от реализации компилятора.
- Директива препроцессора **#define** используется для создания символических констант и макросов.
- Символическая константа — это имя для константы.
- Макрос — это операция, определенная в директиве препроцессора **#define**. Макрос можно определять с аргументами или без них.
- Заменяющий текст для макроса или символической константы — это любой текст, расположенный в строке после идентификатора в директиве **#define**. Если заменяющий текст для макроса или константы не умещается на одной строке, в конце строки следует поместить обратную косую черту (\), которая показывает, что текст продолжается на следующей строке.
- Определения символических констант и макросов можно отменить с помощью директивы препроцессора **#undef**.

- Область действия символической константы или макроса простирается от места их определения до места отмены определения с помощью **#undef**, либо до конца файла.
- Условная компиляция позволяет программисту управлять выполнением директив препроцессора и компиляцией программного кода.
- Условные директивы препроцессора вычисляют значение целочисленного выражения. В директивах препроцессора не допускаются выражения приведения типа, выражения **sizeof** и перечислимые константы.
- Каждая конструкция **#if** заканчивается **#endif**.
- Директивы **#ifdef** и **#ifndef** представляют собой сокращения для **#if defined(*имя*)** и **#if !defined(*имя*)**.
- Условные конструкции препроцессора, проверяющие несколько вариантов, реализуются с помощью директив **#elif** (эквивалента **else if** структуры **if**) и **#else** (эквивалента **else** структуры **if**).
- Директива **#error** выводит на печать зависящее от реализации сообщение, включая указанные в директиве лексемы.
- Директива **#pragma** указывает действие, зависящее от реализации. Указание, не распознанное данным компилятором, игнорируется.
- Операция # производит замену текстовой лексемы, которая преобразуется в строку, заключенную в кавычки. Операция # должна использоваться в макросах с аргументами, поскольку операнд после # ссылается на аргумент макроса.
- Операция ## выполняет конкатенацию двух лексем. Операция ## должна иметь два операнда.
- Директива препроцессора **#line** используется для последовательной нумерации строк исходного кода программы, начиная с числа, заданного константой.
- Существует пять предопределенных символических констант. Константа **__LINE__** — номер текущей обрабатываемой строки исходного кода программы (целая константа). Константа **__FILE__** — имя компилируемого исходного файла (символьная строка). Константа **__DATE__** — дата начала компиляции текущего исходного файла (строка). Константа **__TIME__** — время начала компиляции текущего исходного файла (строка). Константа **__STDC__** — целая константа **1**; предназначена, чтобы показать совместимость данной реализации со стандартом ANSI. Каждая из предопределенных констант начинается и заканчивается двумя символами подчеркивания.
- Макрос **assert**, определенный в заголовочном файле **assert.h**, проверяет значение выражения. Если значение выражения равно **0** (false), то **assert** распечатывает сообщение об ошибке и вызывает функцию **abort** для завершения работы программы.

Терминология

#define
#elif
#else
#endif
#error
#if
#ifdef
#ifndef
#include «filename»
#include <filename>
#line
#pragma
#undef
\ (обратная дробная черта)
символ продолжения
__DATE__
__FILE__
__LINE__
__STDC__
__TIME__
a.out в UNIX
abort
assert
assert.h
stdio.h
stdlib.h

аргумент
директива препроцессора
заголовочные файлы
стандартной библиотеки
заголовочный файл
заменяющий текст
команда **cc** в UNIX
макроопределение
макрос
макрос с аргументами
область действия символической
константы или макроса
отладчик
предопределенные символические
константы
препроцессор С
препроцессорная операция
конкатенации ##
препроцессорная операция
преобразования в строку #
расширение макроса
символическая константа
условная компиляция
условное выполнение директив
препроцессора

Распространенные ошибки программирования

13.1. Забывают заключить аргументы в скобки в заменяющем тексте макроса.

13.2. Вставка условно компилируемых операторов **printf** для целей отладки в места, где С подразумевает только один оператор. В этом случае условно компилируемый фрагмент должен быть включен в составной оператор. Тогда, если программа компилируется с отладочными операторами, поток управления не изменяется.

Хороший стиль программирования

13.1. Использование осмысленных имен для именованных констант делает программу самодокументированной и облегчает ее восприятие.

Советы по повышению эффективности

13.1. Иногда вместо вызова функции можно использовать макрос, который встраивает код в программу до времени исполнения. Это исключает накладные расходы, связанные с вызовом функции.

Упражнения для самоконтроля

13.1. Заполните пробелы в каждом из следующих утверждений.

a) Каждая директива препроцессора должна начинаться с _____.

b) Конструкции условной компиляции можно использовать для проверки нескольких вариантов, если воспользоваться директивами _____ и _____.

c) Директива _____ создает макросы и символические константы.

d) Только _____ символы могут стоять в строке перед директивой препроцессора.

e) Директива _____ отменяет символические константы и макросы.

f) Директивы _____ и _____ — сокращенные нотации для **#if defined(*имя*)** и **#if !defined(*имя*)**.

g) _____ дает возможность программисту управлять выполнением директив препроцессора и компиляцией программного кода.

h) _____ — макрос, выдающий на печать сообщение и прекращающий выполнение программы, если оценка выражения в макросе равна 0.

i) Директива _____ вставляет файл в другой файл.

j) Операция _____ выполняет конкатенацию двух аргументов.

k) Операция _____ преобразует операнд в строку.

l) Символ _____ означает, что заменяющий текст для символической константы или макроса продолжается на следующей строке.

m) Директива _____ нумерует строки исходного кода программы, следующие непосредственно за директивой, начиная с заданного номера.

13.2. Напишите программу, выводящую на печать значения предопределенных символических констант, перечисленных на рис. 13.1.

13.3. Напишите директивы препроцессора, выполняющие каждое из следующих действий.

a) Определите символическую константу **YES**, присвоив ей значение **1**.

b) Определите символическую константу **NO**. присвоив ей значение **0**.

c) Включите в программу заголовочный файл **common.h**. Заголовочный файл находится в том же каталоге, что и компилируемый файл.

d) Пронумеруйте оставшиеся строки в файле, начиная с номера **3000**.

e) Если была определена символическая константа **TRUE**, отмените ее определение, после чего вновь определите ее, присвоив ей значение **1**. Сделайте это, не используя директиву препроцессора **#ifdef**.

f) Если была определена символическая константа **TRUE**, отмените ее определение, после чего вновь определите ее, присвоив ей значение **1**. Сделайте это, используя директиву препроцессора **#ifdef**.

g) Если символическая константа **TRUE** не равна **0**, определите символическую константу **FALSE** равной **0**. В противном случае определите **FALSE** равной **1**.

h) Определите макрос **SQUARE_VOLUME**, вычисляющий объем куба. Макрос должен иметь один аргумент.

Ответы на упражнения для самоконтроля

13.1. a) **#**, b) **#elif**, **#else**, c) **#define**, d) пробельные, e) **#undef**, f) **#ifdef**, **#ifndef**, g) Условная компиляция, h) **assert**, i) **#include**, j) **##**, k) **#**, l) ****, m) **#line**.

13.2.
```
/* Напечатать значения предопределенных макросов */
#include <stdio.h>

main()
{
   printf("__LINE__= %d\n", __LINE__);
   printf("__FILE__= %s\n", __FILE__);
   printf("__DATE__= %s\n", __DATE__);
   printf("__TIME__= %s\n", __TIME__);
   printf("__STDC__= %d\n", __STDC__);
}
```

```
__LINE__ = 5
__FILE__ = macros.c
__DATE__ = Sep 08 1993
__TIME__ = 10:23:47
__STDC__ = 1
```

13.3. a)
```
#define YES 1
```

b)
```
#define NO 0
```

c)
```
#include "common.h"
```

d)
```
#line 3000
```

e)
```
#if defined(TRUE)
   #undef TRUE
   #define TRUE 1
#endif
```

f)
```
#ifdef TRUE
   #undef TRUE
   #define TRUE 1
#endif
```

g)
```
#if TRUE
   #define FALSE 0
#else
   #define FALSE 1
#endif
```

h)
```
#define SQUARE_VOLUME(x)  (x) * (x) * (x)
```

Упражнения

13.4. Напишите программу, которая определяет макрос с одним аргументом для вычисления объема шара. Программа должна вычислять объем шаров с радиусом от одного до десяти, и выводить на печать результаты в виде таблицы. Объем шара вычисляется по следующей формуле:

$(4/3) * \pi * r^3$

где π равно **3.14159**.

13.5. Напишите программу, которая выдает следующее сообщение:

```
The sum of x and y is 13
```

В программе необходимо определить макрос **SUM** с двумя аргументами **x** и **y**, и использовать **SUM** для вывода сообщения.

13.6. Напишите программу, использующую макрос **MINIMUM2** для определения меньшего из двух чисел. Ввод значений должен производиться с клавиатуры.

13.7. Напишите программу, использующую макрос **MINIMUM3** для определения наименьшего из трех чисел. Макрос **MINIMUM3** должен использовать макрос **MINIMUM2**, определенный в упражнении 13.6. Ввод значений должен производиться с клавиатуры.

13.8. Напишите программу, использующую макрос **PRINT** для вывода на печать строки.

13.9. Напишите программу, использующую макрос **PRINTARRAY** для вывода на печать массива целых чисел. Макрос должен получать в качестве аргументов имя массива и число элементов в нем.

13.10. Напишите программу, использующую макрос **SUMARRAY** для суммирования значений численного массива. Макрос должен получать в качестве аргументов имя массива и число элементов в нем.

глава

14

Специальные вопросы



Цели

- Научиться переадресации ввода с клавиатуры на ввод из файла.
- Научиться переадресации экранного вывода в файл.
- Научиться писать функции, использующие списки аргументов переменной длины.
- Научиться обрабатывать аргументы командной строки.
- Научиться присваивать числовым константам конкретный тип данных.
- Научиться использованию временных файлов.
- Научиться программно обрабатывать непредвиденные события.
- Научиться динамически выделять память под массивы.
- Научиться изменять размеры выделенной ранее динамической памяти.

Содержание

14.1. Введение
14.2. Переадресация ввода/вывода в системах UNIX и DOS
14.3. Списки аргументов переменной длины
14.4. Аргументы командной строки
14.5. Замечания относительно компиляции программ из нескольких исходных файлов
14.6. Выход из программы с помощью exit и atexit
14.7. Модификатор типа volatile
14.8. Суффиксы для целых констант и констант с плавающей точкой
14.9. Еще раз о файлах
14.10. Обработка сигналов
14.11. Динамическое выделение памяти: функции calloc и realloc
14.12. Безусловный переход: goto

Резюме • Распространенные ошибки программирования • Советы по переносимости программ • Советы по повышению эффективности • Общие методические замечания • Упражнения для самоконтроля • Ответы на упражнения для самоконтроля • Упражнения

14.1. Введение

В этой главе представлен ряд вопросов, обычно не включаемых во вводные курсы по языку С. Многие из обсуждаемых здесь возможностей рассматриваются в применении к конкретным операционным системам, главным образом UNIX и/или DOS.

14.2. Переадресация ввода/вывода в системах UNIX и DOS

Как правило, устройством ввода данных в программу является клавиатура (стандартный ввод), а устройством вывода — экран монитора (стандартный вывод). В большинстве операционных систем, и в частности в системах UNIX и DOS, можно *переадресовать* ввод так, чтобы он производился не с клавиату-

ры, а из файла, а также переадресовать вывод в файл вместо экрана. Обе формы переадресации можно выполнить без использования средств стандартной библиотеки работы с файлами.

Существует несколько способов переадресации ввода/вывода из командной строки UNIX. Рассмотрим исполняемый файл **sum**, который вводит по одному целые числа и сохраняет значение суммы введенных величин до тех пор, пока не будет установлен индикатор конца файла, после чего распечатывает результат. Обычно пользователь вводит числа с клавиатуры и вводит комбинацию конца файла, чтобы показать, что ввод данных окончен. При переадресации ввода данные, которые необходимо ввести, можно взять из файла. Например, если данные сохранены в файле **input**, командная строка

```
$ sum < input
```

запускает программу **sum**; *символ переадресации ввода (<)* указывает, что для ввода в программу должны использоваться данные из файла **input**. Переадресация ввода в системе DOS производится точно так же.

Отметим, что **$** — приглашение командной строки UNIX (некоторые UNIX-системы используют в качестве приглашения %). Обучающиеся часто с трудом воспринимают тот факт, что переключение является функцией операционной системы, а не еще одной возможностью С.

Вторым способом переназначения ввода является использование *конвейера*. Конвейер (|) вызывает переадресацию вывода одной программы на ввод другой программы. Предположим, программа **random** выводит ряд случайных чисел; вывод программы **random** можно направить прямо в программу **sum**, используя командную строку UNIX

```
$ random | sum
```

Это приведет к тому, что будет вычислена сумма чисел, выданных **random**. Конвейер можно использовать в UNIX и DOS.

Чтобы перенаправить вывод из программы в файл, используется *символа переадресации вывода (>)* (это возможно как в UNIX, так и в DOS). Например, для переназначения вывода из программы **random** в файл **out** воспользуйтесь следующей командной строкой:

```
$ random > out
```

И наконец, вывод из программы можно добавить в конец существующего файла с помощью *символа присоединения вывода (>>)* (этот символ используется и в UNIX, и в DOS). Например, чтобы добавить вывод из программы **random** в файл **out**, созданный с помощью предыдущей командной строки, воспользуйтесь командой

```
$ random >> out
```

14.3. Списки аргументов переменной длины

Существует возможность создать функцию, число аргументов которой не фиксированно. В большинстве программ, представленных в этой книге, использовалась стандартная библиотечная функция **printf**, которая, как вы знаете, принимает переменное число аргументов. Как минимум, **printf** должна получить строку в качестве первого аргумента, но, кроме того, **printf** может принять любое количество дополнительных аргументов. Прототип функции **printf** выглядит следующим образом:

```
int print(const char *format, ...);
```

Многоточие (...) в прототипе функции означает, что функция получает переменное число аргументов любого типа. Заметим, что многоточие должно всегда находиться в конце списка параметров.

Макросы и определения *заголовочного файла переменных аргументов* **stdarg.h** (рис.14.1) предоставляют программисту средства, необходимые для построения функций со списком аргументов переменной длины. Программа на рис. 14.2 демонстрирует функцию **average**, получающую произвольное количество аргументов. Первый аргумент **average** — это всегда число усредняемых значений.

Идентификатор	Объяснение
va_list	Тип, предназначающийся для хранения информации, необходимой макросам **va_start**, **va_arg** и **va_end**. Чтобы получить доступ к аргументам в списке переменной длины, необходимо объявить объект типа **va_list**.
va_start	Макрос, который вызывается перед обращением к аргументам списка переменной длины. Макрос инициализирует объект, объявленный с помощью **va_list** для использования макросами **va_arg** и **va_end**.
va_arg	Макрос, расширяющийся до выражения со значением и типом следующего аргумента в списке переменной длины. Каждый вызов **va_arg** изменяет объект, объявленный с помощью **va_list** так, что объект указывает на следующий аргумент списка.
va_end	Макрос обеспечивает нормальный возврат из функции, на список аргументов которой ссылался макрос **va_start**.

Рис. 14.1. Тип и макросы, определенные в заголовочном файле **stdarg.h**

Функция **average** применяет все определения и макросы заголовочного файла **stdarg.h**. Объект **ap**, типа **va_list**, используется макросами **va_start**, **va_arg** и **va_end** для обработки списка аргументов переменной длины функции **average**. Функция начинается вызовом макроса **va_start**, инициализирующего объект **ap**. Макрос получает два аргумента: объект **ap** и идентификатор самого правого параметра в списке перед многоточием. В данном случае это **i** (**va_start** нужен идентификатор **i** для определения того, где начинается список аргументов переменной длины). Затем функция **average** последовательно складывает аргументы из списка в переменной **total**. Прибавляемое к **total** значение извлекается из списка аргументов вызовом макроса **va_arg**. Макрос **va_arg** получает два аргумента: объект **ap** и тип значения, ожидаемого в списке аргументов функции. В данном случае это **double**. Макрос возвращает значение аргумента. Функция **average** вызывает макрос **va_end** с объектом **ap** для упрощения нормального возврата из **average** в **main**. И наконец, вычисляется среднее и его значение возвращается в **main**.

Распространенная ошибка программирования 14.1

Многоточие в середине списка параметров функции. Многоточие может находиться только в конце списка параметров.

```
/* Использование списка аргументов переменной длины */
#include <stdio.h>
#include <stdarg.h>

double average(int, ...);

main()
{
   double w = 37.5, x = 22.5, y = 1.7, z = 10.2;

   printf("%s%.1f\n%s%.1f\n%s%.1f\n%s%.1f\n\n",
          "w = ", w, "x = ", x, "y = ", y, "z = ", z);
   printf("%s%.3f\n%s%.3f\n%s%.3f\n",
          "The average of w and x is ",
          average(2, w, x),
          "The average of w, x, and y is ",
          average(3, w, x, y),
          "The average of w, x, y, and z is ",
          average(4, w, x, y, z));

   return 0;
}

double average(int i, ...)
{
   double total = 0;
   int j;
   va_list ap;
   va_start(ap, i);

   for (j = 1; j <= i; j++)
      total += va_arg(ap, double);

   va_end(ap);
   return total / i;
}
```

```
w = 37.5
x = 22.5
y = 1.7
z = 10.2

The average of w and x is 30.000
The average of w, x and y is 20.567
The average of w, x, y, and z is 17.975
```

Рис. 14.2. Использование списка аргументов переменной длины

У читателя может возникнуть резонный вопрос, как функции **printf** и **scanf** узнают, какой тип использовать в каждом макросе **va_arg**. Ответ состоит в том, что **printf** и **scanf** просматривают спецификации преобразования в управляющей строке формата для определения типа следующего аргумента, который будет обрабатываться.

14.4. Аргументы командной строки

Во многих системах, в том числе DOS и UNIX, существует возможность передавать аргументы функции **main** из командной строки посредством включения параметров **int argc** и **char *argv[]** в список параметров **main**. Параметр **argc** получает число аргументов в командной строке. Параметр **argv** — массив строк, в котором сохраняются имеющиеся в командной строке аргументы. Обычное использование аргументов командной строки включает вывод аргументов на печать, передачу опций и передачу программе имен файлов.

Программа на рис. 14.3 посимвольно копирует один файл в другой. Допустим, исполняемый файл данной программы называется **copy**. Типичная командная строка для программы **copy** в системе UNIX выглядит так:

```
$ copy input output
```

```
/* Использование аргументов командной строки */
#include <stdio.h>

main(int argc, char *argv[])
{
   FILE *inFilePtr, *outFilePtr;
   int c;

   if (argc !=3)
      printf("Usage: copy infile outfile\n");
   else
      if ((inFilePtr = fopen(argv[1], "r")) != NULL)

         if ((outFilePtr = fopen(argv[2], "w")) != NULL)

            while ((c = fgetc(inFilePtr)) !=EOF)
               fputc(c, outFilePtr);

         else
            printf("File \"%s\" could not be opend\n", argv[2]);

      else
         printf("File \"%s\" could not be opened\n", argv[1]);
   return 0;
}
```

Рис. 14.3. Использование аргументов командной строки

Данная командная строка показывает, что файл **input** будет скопирован в файл **output**. Во время выполнения программы, если **argc** не равен **3** (**copy** считается одним из аргументов), она выдает сообщение об ошибке и завершается. В противном случае массив **argv** будет содержать строки "**copy**", "**input**" и "**output**". Второй и третий аргументы командной строки воспринимаются программой в качестве имен файлов. Файлы открываются при помощи функции **fopen**. В случае успешного открытия файлов символы считываются из **input** и записываются в **output** до тех пор, пока для файла **input** не будет установлен индикатор конца файла. После этого программа завершается. Результатом является создание точной копии файла **input**. Заметим, что не на всех компьютерных системах работать с аргументами командной строки так же

просто, как в UNIX и DOS. Например, в системах Macintosh и VMS требуются специальные установки для обработки аргументов командной строки. Чтобы получить более полную информацию об использовании аргументов командной строки, воспользуйтесь руководством по вашей системе.

14.5. Замечания относительно компиляции программ из нескольких исходных файлов

Как было отмечено ранее, можно создавать программы, которые состоят из нескольких исходных файлов (см. главу 16, «Классы»). Существует несколько правил, которые не следует забывать при создании многофайловых программ. Например, определение функции должно целиком находиться в одном файле, его нельзя разделить на несколько файлов.

В главе 5 нами были введены понятия классов памяти и области действия. Мы узнали, что переменные, объявленные вне определения любой из функций, по умолчанию принадлежат к статическому классу памяти, и на них ссылаются как на глобальные переменные. Глобальные переменные доступны любой функции, определенной в том же файле после объявления этих переменных. Глобальные переменные также доступны и функциям в других файлах, однако они должны объявляться в каждом из них. Например, если мы определяем глобальную переменную целого типа **flag** в одном файле, и ссылаемся на нее в другом файле, то последний должен содержать объявление

```
extern int flag;
```

Только после этого можно будет обращаться к переменной. В этом объявлении спецификатор **extern** указывает компилятору, что переменная **flag** определена или в другом файле, или в этом же, но позже. Компилятор сообщает компоновщику, что в файле имеется неразрешенная ссылка на переменную **flag** (компилятор не знает, в каком месте определена переменная **flag**, поэтому предоставляет ее поиск компоновщику). Если компоновщик не найдет определение **flag**, возникнет ошибка компоновки и не будет создано исполняемого файла. Если же соответствующее глобальное определение будет обнаружено, компоновщик разрешит ссылки, указав, где находится **flag**.

Совет по повышению эффективности 14.1

Глобальные переменные увеличивают производительность, потому что они доступны напрямую из любой функции — дополнительные затраты на передачу данных функциям, таким образом, устраняются.

Общее методическое замечание 14.1

Следует избегать применения глобальных переменных за исключением случаев, когда быстродействие приложения является критическим фактором, поскольку они нарушают принцип минимума привилегий.

Так же как объявление **extern** позволяет объявлять глобальные переменные в других файлах программы, прототипы функций могут расширить область действия функций за пределы файла, в котором они определены (спецификатор **extern** в прототипе функции не обязателен). Для этого необходимо

включить прототип функции в каждый файл, в котором функция вызывается, и компилировать файлы совместно (см. раздел 13.2). Прототипы функций указывают компилятору, что данная функция или определена в другом файле, или будет определена позже в этом. И снова компилятор не пытается разрешить ссылки на такую функцию, эта обязанность поручается компоновщику. Если компоновщик не сумеет обнаружить соответствующее определение функции, возникает ошибка.

Как пример использования прототипа функции для расширения ее области действия рассмотрите любую программу, содержащую директиву препроцессора **#include <stdio.h>**. Этим директива включает в файл прототипы таких функций, как **printf** и **scanf**. Другие функции в файле могут использовать **printf** и **scanf** для выполнения своих задач. Функции **printf** и **scanf** определены отдельно, и нам нет необходимости знать, где они определены. Мы просто постоянно используем их код в своих программах. Компоновщик разрешает наши ссылки на эти функции автоматически. Этот процесс позволяет нам использовать функции стандартной библиотеки.

Общее методическое замечание 14.2

Создание программ из нескольких исходных файлов способствует повторному использованию кода и систематизированному конструированию программного обеспечения. Функции могут использоваться во многих приложениях. В таких случаях следует сохранять эти функции в отдельных исходных файлах, и каждый исходный файл должен иметь соответствующий ему заголовок, содержащий прототипы функций. Это позволяет программистам использовать в различных программах один и тот же код, включая в них соответствующий заголовочный файл и компилируя эти программы вместе с уже существующим исходным файлом.

Совет по переносимости программ 14.1

Некоторые операционные системы не поддерживают имена глобальных переменных или функций длиной более шести символов. Об этом не следует забывать при разработке программ, предназначенных для работы на различных машинах.

Существует возможность ограничить область действия глобальной переменной или функции заданным файлом. Спецификатор класса памяти **static**, будучи применен к глобальной переменной или функции, предотвращает обращение к ней из любой функции, не определенной в том же самом файле. Иногда это называют *внутренней компоновкой*. Глобальные переменные и функции, определение которых не предваряется **static**, допускают *внешнюю компоновку*, и другие файлы могут получить к ним доступ, если содержат соответствующие объявления и/или прототипы функций.

Объявление глобальной переменной

```
static float pi = 3.14159;
```

создает переменную **pi** типа **float**, присваивает ей значение **3.14159** и показывает, что **pi** известна только функциям файла, в котором она определена.

Спецификатор **static** обычно используется со вспомогательными функциями, которые вызываются только функциями отдельного файла. Если функция не используется за пределами отдельного файла, то следует применять принцип минимальных привилегий, объявляя ее как **static**. Если функция определена до того, как она используется в файле, **static** должен применяться к определению функции. В противном случае **static** применяется к ее прототипу.

При разработке больших программ, состоящих из нескольких исходных файлов, компиляция программы становится утомительным делом; если в одном из файлов сделаны небольшие изменения, приходится перекомпилировать всю программу. Многие системы снабжены специальными утилитами, которые перекомпилируют только модифицированные файлы программы. В UNIX-системах подобная утилита называется **make**. Утилита **make** читает файл с именем **makefile**, который содержит инструкции по компиляции и компоновке программы. Такие системы, как Borland C++ и Microsoft C/C++ 7.0 для PC, также имеют утилиту **make**. Более подробную информацию относительно этой утилиты можно получить, прочитав руководство по вашей системе.

14.6. Выход из программы с помощью exit и atexit

Библиотека общего назначения (**stdlib.h**) предусматривает также иные способы выхода из выполняющейся программы, нежели традиционный возврат из функции **main**. Функция **exit** вызывает завершение работы программы, как если бы она выполнилась нормально. Эта функция часто используется для того, чтобы прекратить выполнение, когда при вводе обнаружена ошибка, или невозможно открыть файл, который должна обрабатывать программа. Функция **atexit** *регистрирует* функцию, которая будет вызвана при успешном завершении программы, т.е. если управление достигает конца функции **main** или вызывается функция **exit**.

Функции **atexit** в качестве аргумента передается указатель на функцию (т.е. имя функции). Функция, вызываемая при завершении программы, не может иметь аргументов и не может возвращать значения. Можно зарегистрировать до 32 функций, которые должны выполняться при завершении программы.

Функция **exit** принимает один аргумент. Аргумент, как правило — символическая константа **EXIT_SUCCESS** или **EXIT_FAILURE**. Если **exit** вызвана с **EXIT_SUCCESS**, окружению возвращается значение успешного завершения, зависящее от реализации. Если **exit** вызвана с **EXIT_FAILURE**, окружению возвращается значение неудачного завершения. Когда вызывается функция **exit**, все функции, предварительно зарегистрированные с помощью **atexit**, вызываются в порядке, обратном их регистрации, все связанные с программой потоки сбрасываются и закрываются, а управление возвращается системе. Программа на рис. 14.4 тестирует функции **exit** и **atexit**. Программа предлагает пользователю определить, завершать ли программу с помощью **exit** или по достижении конца функции **main**. Отметим, что функция **print** выполняется при завершении программы в любом случае.

14.7. Модификатор типа volatile

В главах 6 и 7 мы ввели модификатор типа **const**. ANSI C, кроме того, поддерживает модификатор типа **volatile**. В стандарте ANSI (An90) сказано, что когда для спецификации типа используется **volatile**, природа доступа к объекту этого типа зависит от реализации системы. Керниган и Ричи (Ke88) указывают, что модификатор **volatile** должен использоваться для запрещения разного рода оптимизаций.

```
/* Применение функций exit и atexit */
#include <stdio.h>
#include <stdlib.h>

void print(void);

main()
{
   int answer;

   atexit(print);        /* регистрация функции print */
   printf("Enter 1 to terminate program with function exit\n"
          "Enter 2 to terminate program normally\n");
   scanf("%d", &answer);

   if (answer == 1) {
      printf("\nTerminating program with function exit\n");
      exit(EXIT_SUCCESS);
   }

   printf("\nTerminating program by reaching the end of main\n");
   return 0;
}

void print(void)
{
   printf("Executing function print at program termination\n"
          "Program terminated\n");
}
```

```
Enter 1 to terminate program with function exit
Enter 2 to terminate program normally
: 1

Terminating program with function exit
Executing function print at program termination
Program terminated

Enter 1 to terminate program with function exit
Enter 2 to terminate program normally
: 2

Terminating program by reaching the end of main
Executing function print at program termination
Program terminated
```

Рис. 14.4. Применение функций **exit** и **atexit**

14.8. Суффиксы для целых констант и констант с плавающей точкой

В C предусмотрены суффиксы для целых констант и констант с плавающей точкой. Целыми суффиксами являются: **u** или **U** для целого **unsigned**, **l** или **L** для целого типа **long**, и **ul** или **UL** для целого типа **unsigned long**. Следующие константы имеют соответственно тип **unsigned**, **long**, и **unsigned long**:

```
174u
8358L
28373ul
```

Если целая константа не имеет суффикса, ее тип определяется ближайшим типом, способным хранить значение такой величины (сначала **int**, потом **long int**, и, наконец **unsigned long int**)

Суффиксы констант с плавающей точкой следующие: **f** или **F** для **float**, и **l** или **L** для **long double**. Следующие константы имеют соответственно тип **float** и **long double**:

```
1.28f
3.14159L
```

Константы с плавающей точкой без суффикса автоматически получают тип **double**.

14.9. Еще раз о файлах

В главе 11 обсуждалась возможность обработки текстовых файлов с последовательным и произвольным доступом. Язык С также предусматривает возможность обработки двоичных файлов, но некоторые компьютерные системы их не поддерживают. Если двоичные файлы не поддерживаются и файл открыт в двоичном режиме (рис. 14.5), файл будет обрабатываться как текстовый. Двоичные файлы должны использоваться вместо текстовых, только когда существуют жесткие требования к скорости, памяти и/или условия совместимости требуют применения двоичных файлов. В противном случае текстовые файлы во всех отношениях предпочтительней благодаря присущей им переносимости, а также возможности использовать другие стандартные инструменты для просмотра и работы с текстовыми файлами данных.

Режим	Описание
rb	Открывает двоичный файл для чтения.
wb	Создает двоичный файл для записи. Если файл уже существует, удаляет текущее содержимое.
ab	Добавление; открывает или создает двоичный файл для записи в конец файла.
rb+	Открывает двоичный файл для обновления (чтение и запись).
wb+	Создает двоичный файл для обновления. Если файл уже существует, удаляет его текущее содержимое.
ab+	Добавление; открывает или создает двоичный файл для обновления; все записи производятся в конец файла.

Рис. 14.5. Режимы открытия двоичных файлов

В дополнение к функциям обработки файлов, которые обсуждались в главе 11, в стандартной библиотеке имеется также функция **tmpfile**, которая открывает временный файл в режиме "**wb+**". Хотя это режим для двоичного файла, некоторые системы обрабатывают временные файлы как текстовые. Временный файл существует до тех пор, пока не будет закрыт с помощью **fclose**, или пока не завершится выполнение программы.

Совет по повышению эффективности 14.2

Рассмотрите возможность использования двоичных файлов вместо текстовых в приложениях, требующих высокой производительности.

Совет по переносимости программ 14.2

Используйте текстовые файлы, когда пишете программы, которые планируется переносить на другие системы.

Программа на рис. 14.6 заменяет в файле символы табуляции на пробелы. Пользователю предлагается ввести имя файла, который будет модифицироваться. Если введенный пользователем файл и временный файл успешно открыты, программа считывает символы из файла, который необходимо обработать, и записывает их во временный файл. Если считывается символ табуляции (`'\t'`), он заменяется пробелом, который и записывается во временный файл. Когда достигается конец исходного файла, указатели позиции каждого из файлов устанавливаются на начало вызовом **rewind**. После этого временный файл копируется посимвольно в исходный файл. Программа выводит на печать первоначальный файл по ходу копирования символов из него во временный файл, и выводит на печать временный файл по ходу копирования его символов в первоначальный, чтобы подтвердить, что символы были записаны.

```
/* Использование временных файлов */
#include <stdio.h>

main()
{
   FILE *filePtr, *tempFilePtr;
   int c;
   char fileName[30];

   printf("This prgram changes tabs to spaces.\n"
          "Enter a file to be modified: ");
   scanf("%s", fileName);

   if ( ( filePtr = fopen(fileName, "r+") ) != NULL)

      if ( ( tempFilePtr = tmpfile() ) != NULL) {
         printf("\nThe file before modification is:\n");
         while ( ( c = getc(filePtr) ) != EOF) {
            putchar(c);
            putc(c == '\t' ? ' ': c, tempFilePtr);
         }

         rewind(tempFilePtr);
         rewind(filePtr);
         printf("\n\nThe file after modification is:\n");

         while ( ( c = getc(tempFilePtr) ) != EOF) {
            putchar(c);
            putc(c, filePtr);
         }
```

Рис. 14.6. Использование временных файлов (часть 1 из 2)

```
      }
      else
         printf("Unable to open temporary file\n");

   else
      printf("unable to open %s\n", fileName);

   return 0;
}
```

```
This program changes tabs to spaces.
Enter a file to be modified: data

The file before modification is:
0       1       2       3       4
        5       6       7       8       9

The file after modification is:
0 1 2 3 4
  5 6 7 8 9
```

Рис. 14.6. Использование временных файлов (часть 2 из 2)

14.10. Обработка сигналов

Непредвиденное событие, или *сигнал*, может привести к тому, что программа будет завершена преждевременно. Вот лишь некоторые из непредвиденных событий: *прерывания* (ввод **<ctrl>c** в UNIX или DOS), *недопустимые инструкции*, *нарушение сегментации*, *запрос на завершение от операционной системы*, и *исключительные ситуации с плавающей точкой* (деление на ноль или переполнение). Библиотека обработки сигналов обеспечивает возможность *перехвата* непредвиденных событий с помощью функции **signal**. Функция **signal** получает два аргумента: целое число — номер сигнала, и указатель на функцию обработки сигналов. Сигналы могут генерироваться функцией **raise**, которая принимает целое число — номер сигнала в качестве аргумента. На рис. 14.7 представлена сводная таблица стандартных сигналов, определенных в заголовочном файле **signal.h**. Программа на рис. 14.8 демонстрирует функции **signal** и **raise** в действии.

Сигнал	Пояснение
SIGABRT	Аварийное завершение программы (например, вызов функции **abort**).
SIGFPE	Ошибочная арифметическая операция, например деление на ноль или операция, вызывающая переполнение.
SIGILL	Обнаружение недопустимой инструкции.
SIGINT	Получение интерактивного сигнала прерывания.
SIGSEGV	Некорректный доступ к памяти.
SIGTERM	Запрос на завершение, направленный программе.

Рис. 14.7 Сигналы, определенные в заголовочном файле **signal.h**

```
/* Обработка сигналов */
#include <stdio.h>
#include <signal.h>
#include <stdlib.h>
#include <time.h>

void signal_handler(int);

main()
{
   int i, x;

   signal(SIGINT, signal_handler);
   srand(clock());

   for (i = 1; i <= 100; I++){
      x = 1 + rand() % 50;

      if (x == 25)
         raise(SIGINT);

      printf("%4d", I);

      if (i % 10 == 0)
         printf("\n");
   }

   return 0;
}

void signal_handler(int signalValue)
{
   int response;

   printf("%s%d%s\n%s",
          "\nInterrupt signal (", signalValue, ") received.",
          "Do you wish to continue (1 = yes or 2 = no)? ");
   scanf("%d", &response);
   while (response != 1 && response != 2) {
      printf("(1 = yes or 2 = no)? ");
      scanf("%d", &response);
   }

   if (response == 1)
      signal(SIGINT, signal_handler);
   else
      exit(EXIT_SUCCESS);
}
```

Рис. 14.8 Использование обработки сигналов (часть 1 из 2)

Программа на рис. 14.8 вызывает функцию **signal**, чтобы перехватить интерактивный сигнал (**SIGINT**). Программа начинается вызовом **signal** с **SIGINT** и указателем на функцию **signal_handler** (помните, что имя функции является указателем на ее начало) в качестве аргументов. Когда генерируется сигнал типа **SIGINT**, управление передается функции **signal_handler**, выдается сооб-

щение на печать, и пользователь получает возможность продолжить нормальное выполнение программы. Если пользователь хочет продолжить выполнение, обработчик сигналов вновь инициализируется повторным вызовом **signal** (некоторые системы требуют, чтобы обработчик сигналов был повторно инициализирован) и управление передается в ту точку программы, в которой был обнаружен сигнал. В этой программе для моделирования интерактивного сигнала используется функция **raise**. Выбирается произвольное число между **1** и **50**. Если число равно **25**, то вызывается **raise** для генерации сигнала. Обычно интерактивные сигналы инициируются извне. Например, ввод **<ctrl> c** во время выполнения программы в системах UNIX или DOS генерирует интерактивный сигнал, который прерывает выполнение программы. Обработку сигналов можно использовать для перехвата интерактивных сигналов и предотвращения прерывания выполнения программы.

```
  1   2   3   4   5   6   7   8   9   10
 11  12  13  14  15  16  17  18  19   20
 21  22  23  24  25  26  27  28  29   30
 31  32  33  34  35  36  37  38  39   40
 41  42  43  44  45  46  47  48  49   50
 51  52  53  54  55  56  57  58  59   60
 61  62  63  64  65  66  67  68  69   70
 71  72  73  74  75  76  77  78  79   80
 81  82  83  94  95  86  87  88
Interrupt signal (4) received.
Do you wish to continue (1 = yes or 2 = no)? 1
 89  90
 91  92  93  94  95  96  97  98  99   100
```

Рис. 14.8 Использование обработки сигналов (часть 2 из 2)

14.11. Динамическое выделение памяти: функции calloc и realloc

В главе 12, «Структуры данных», говорилось о динамическом выделении памяти с помощью функции **malloc**. Как мы тогда констатировали, массивы лучше, чем связанные списки, подходят для быстрой сортировки, поиска и доступа к данным. Однако массивы, как правило, являются *статическими структурами данных*. Библиотека общего назначения (**stdlib.h**) поддерживает еще две функции для динамического выделения памяти: **calloc** и **realloc**. С помощью этих функций можно создавать и модифицировать *динамические массивы*. Как показано в главе 7, «Указатели», указатель на массив может быть индексирован как массив. Таким образом, указателем на непрерывный блок памяти, созданным **calloc**, можно манипулировать как массивом. Функция **calloc** динамически выделяет память для массива. Прототип **calloc** выглядит следующим образом:

```
void *calloc(size_t nmemb, size_t size);
```

Функция принимает два аргумента: число элементов (**nmemb**) и размер каждого элемента (**size**), и инициализирует элементы массива нулями. Функция возвращает указатель на выделенную память, или **NULL**, если память не может быть выделена.

Функция **realloc** изменяет размер объекта, выделенного в результате предыдущего вызова **malloc**, **calloc** или **realloc**. Содержимое первоначального объекта сохраняется при условии, что размер вновь выделяемой памяти больше, чем первоначальный. В противном случае содержимое остается неизменным лишь в пределах размера нового объекта. Прототип **realloc** выглядит следующим образом:

```
void *realloc(void *ptr, size_t size);
```

Функция **realloc** принимает два аргумента: указатель на первоначальный объект (**ptr**) и новый размер объекта (**size**). Если **ptr** равен **NULL**, **realloc** работает аналогично **malloc**. Если **size** равен **0**, а **ptr** не **NULL**, память, выделенная для объекта, освобождается. В противном случае, если **ptr** не **NULL** и размер больше нуля, **realloc** пытается выделить для объекта новый блок памяти. Если новую память выделить не удается, объект, на который указывает **ptr**, не изменяется. Функция **realloc** возвращает либо указатель на вновь выделенную память, либо **NULL**.

14.12. Безусловный переход: goto

На протяжении всей книги мы постоянно подчеркивали важность использования методов структурного программирования, способствующих созданию надежного программного обеспечения, которое легко отлаживать, сопровождать и модифицировать. В некоторых случаях производительность оказывается важнее строгой приверженности канонам структурного программирования. В этом случае допустимо использовать и некоторые неструктурные методы. Например, мы можем использовать **break**, чтобы прервать выполнение структуры повторения до того, как условие продолжения цикла станет ложным. Это исключает ненужные итерации, если задача выполнена до того, как произойдет окончание цикла.

Другим примером неструктурного программирования является применение *оператора* ***goto*** — безусловного перехода. Результатом исполнения **goto** является передача управления первому оператору программы после *метки*, указанной в операторе **goto**. Метка представляет собой идентификатор, за которым следует двоеточие. Метка должна находиться в той же самой функции, что и оператор **goto**, который на нее ссылается. Программа на рис. 14.9 использует операторы **goto**, чтобы десять раз выполнить цикл и при этом каждый раз выводить на печать значение счетчика. После инициализации **count** значением **1** программа проверяет **count**, определяя, не больше ли он **10** (метка **start** пропускается, поскольку метка не выполняет никаких действий). Если больше, то управление передается первому оператору, расположенному за меткой **end**. В противном случае значение **count** выводится на печать и увеличивается на единицу, а управление передается первому оператору после метки **start**.

В главе 3 мы констатировали, что только три управляющих структуры необходимы для написания любой программы: последовательная, выбора и повторения. Следуя правилам структурного программирования, можно создать управляющие структуры с многочисленными уровнями вложенности, из которых очень трудно эффективно выйти. Некоторые программисты используют в таких ситуациях операторы **goto**, как средство быстрого выхода из структуры с многократным вложением. Это избавляет от необходимости проверять многочисленные условия, чтобы выйти из управляющей структуры.

Совет по повышению эффективности 14.3

Оператор **goto** может использоваться для эффективного выхода из глубоких уровней вложенных управляющих структур.

Общее методическое замечание 14.3

Оператор **goto** должен использоваться только в приложениях, ориентированных на высокую производительность. Оператор **goto** не является структурным и программы с этим оператором, как правило, труднее отлаживать, сопровождать и модифицировать.

```
/* Применение goto */
#include <stdio.h>

main()
{
   int count = 1;

   start:                  /* метка */
      if (count >10)
         goto end;

      printf("%d ", count);
      ++count;
      goto start;

   end:                    /* метка */
      putchar('\n');

   return 0;
}
```

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

Рис. 14.9 Использование **goto**

Резюме

- Многие операционные системы, и в частности UNIX и DOS, позволяют переадресовать потоки ввода/вывода программы.
- В системах UNIX и DOS ввод переназначается в командной строке с использованием символа переадресации ввода (<) или конвейера (|).
- В системах UNIX и DOS вывод переназначается в командной строке с использованием символа переадресации вывода (>) или символа присоединения вывода (>>). Применение символа переадресации вывода просто сохраняет выводимые программой данные в файле, а символ присоединения вывода добавляет выводимые программой данные в конец указанного файла.
- Макросы и определения заголовочного файла **stdarg.h** предоставляют программисту средства, необходимые для построения функций со списком аргументов переменной длины.

- Многоточие (...) в прототипе функции указывает на переменное количество аргументов.
- Тип **va_list** предназначается для хранения информации, необходимой макросам **va_start**, **va_arg** и **va_end**. Чтобы получить доступ к аргументам в списке переменной длины, необходимо объявить объект типа **va_list**.
- Макрос **va_start** вызывается перед обращением к аргументам списка переменной длины. Макрос инициализирует объект, объявленный как **va_list**, для использования макросами **va_arg** и **va_end**.
- Макрос **va_arg** расширяется до выражения со значением и типом следующего аргумента в списке переменной длины. Каждый вызов **va_arg** изменяет объект, объявленный с помощью **va_list** так, что объект указывает на следующий аргумент списка.
- Макрос **va_end** упрощает нормальный возврат из функции, на список аргументов переменной длины которой ссылался макрос **va_start**.
- Во многих системах, в частности DOS и UNIX, существует возможность передавать аргументы функции **main** из командной строки посредством включения параметров **int argc** и **char *argv[]** в список параметров **main**. Параметр **argc** передает число аргументов в командной строке. Параметр **argv** — массив строк, в котором сохраняются имеющиеся в командной строке аргументы.
- Определение функции должно целиком находиться в одном файле, его нельзя разделить на несколько файлов.
- Глобальные переменные должны быть объявлены в каждом из файлов, где они используются.
- Прототипы функций могут расширить область действия функций за пределы файла, в котором они определены (спецификатор **extern** в прототипе функции не обязателен). Для этого необходимо включить прототип функции в каждый файл, в котором функция вызывается, и компилировать файлы совместно.
- Спецификатор класса памяти **static**, будучи примененным к глобальной переменной или функции, предотвращает ее использование любой функцией, не определенной в том же самом файле. Иногда это называют внутренней компоновкой. Глобальные переменные и функции, определение которых не предваряется **static**, допускают внешнюю компоновку; другие файлы могут получить к ним доступ, если содержат соответствующие объявления и/или прототипы функций.
- Спецификатор **static** обычно используется со вспомогательными функциями, которые вызываются только функциями отдельного файла. Если функция не используется за пределами того файла, где она определяется, то следует применять принцип минимальных привилегий, объявляя ее как **static**.
- При разработке больших программ из нескольких исходных файлов компиляция программы становится утомительным делом, если в одном из файлов сделаны небольшие изменения, а приходится перекомпилировать всю программу. Многие системы снабжены специальными утилитами, которые перекомпилируют только модифицированные файлы

программы. В UNIX системах подобная утилита называется **make**. Утилита **make** читает файл с именем **makefile**, который содержит инструкции по компиляции и компоновке программы.

- Функция **exit** вызывает завершение работы программы, как если бы она была выполнена нормально.
- Функция **atexit** регистрирует в программе функцию, которую необходимо вызвать при успешном завершении программы, т.е. если управление достигает конца функции **main** или была вызвана функция **exit**.
- Функции **atexit** в качестве аргумента передается указатель на функцию (т.е. имя функции). Функция, вызываемая при завершении программы, не может иметь аргументов и не может возвращать значения. Можно зарегистрировать до 32 функций, выполняющихся при завершении программы.
- Функция **exit** принимает один аргумент. Аргумент, как правило, — символическая константа **EXIT_SUCCESS** или константа **EXIT_FAILURE**. Если **exit** вызвана с **EXIT_SUCCESS**, окружению возвращается значение успешного завершения, зависящее от реализации. Если **exit** вызвана с **EXIT_FAILURE**, окружению возвращается значение неудачного завершения.
- Когда вызывается функция **exit**, все функции, предварительно зарегистрированные с помощью **atexit**, вызываются в порядке, обратном их регистрации, все связанные с программой потоки сбрасываются и закрываются, а управление возвращается системе.
- В стандарте ANSI (An90) сказано, что когда для спецификации типа используется **volatile**, характер доступа к объекту этого типа зависит от реализации. Керниган и Ричи (Ke88) указывают, что модификатор **volatile** должен использоваться для запрещения разного рода оптимизаций.
- В C предусмотрены суффиксы для целых констант и констант с плавающей точкой. Целыми суффиксами являются: **u** или **U** для целого **unsigned**, **l** или **L** для целого типа **long**, и **ul** или **UL** для целого типа **unsigned long**. Если целая константа не имеет суффикса, для нее принимается ближайший тип, способный хранить значение такой величины (сначала **int**, потом **long int**, и наконец **unsigned long int**) Суффиксы констант с плавающей точкой следующие: **f** или **F** для **float**, и **l** или **L** для **long double**. Константы с плавающей точкой без суффикса автоматически получают тип **double**.
- Язык C предусматривает возможность обработки двоичных файлов, но некоторые компьютерные системы их не поддерживают. Если двоичные файлы не поддерживаются и файл открыт в двоичном режиме, он будет обрабатываться как текстовый.
- Функция **tmpfile** открывает временный файл в режиме "**wb+**". Хотя это режим для двоичного файла, некоторые системы обрабатывают временные файлы как текстовые. Временный файл существует до тех пор, пока не будет закрыт с помощью **fclose**, или пока не завершится выполнение программы.

- Библиотека обработки сигналов обеспечивает возможность *перехвата* непредвиденных событий с помощью функции **signal**. Функция **signal** получает два аргумента: целое число — номер сигнала, и указатель на функцию обработки.
- Сигналы могут возбуждаться функцией **raise**, которая принимает целое число — номер сигнала в качестве аргумента.
- Библиотека общего назначения (**stdlib.h**) поддерживает еще две функции для динамического выделения памяти: **calloc** и **realloc**. С помощью этих функций можно создавать и модифицировать динамические массивы.
- Функция **calloc** получает два аргумента: число элементов (**nmemb**) и размер каждого элемента (**size**) и инициализирует элементы массива нулями. Функция возвращает указатель на выделенную память, или **NULL**, если память не выделена.
- Функция **realloc** изменяет размер объекта, выделенного в результате предыдущего вызова **malloc**, **calloc** или **realloc**. Содержимое первоначального объекта сохраняется при условии, что размер выделяемой памяти больше первоначального.
- Функция **realloc** принимает два аргумента: указатель на первоначальный объект (**ptr**) и новый размер объекта (**size**). Если **ptr** равен **NULL**, **realloc** работает аналогично **malloc**. Если **size** равен **0**, а **ptr** не **NULL**, память, выделенная для объекта, освобождается. В противном случае, если **ptr** не **NULL** и размер больше нуля, **realloc** пытается выделить для объекта новый блок памяти. Если новую память выделить не удается, объект, на который указывает **ptr**, не изменяется. Функция **realloc** возвращает или указатель на вновь выделенную память, или **NULL**.
- Результатом исполнения **goto** является передача управления первому оператору программы после метки, указанной в операторе **goto**.
- Метка представляет собой идентификатор, за которым следует двоеточие. Метка должна находиться в той же самой функции, что и оператор **goto**, который на нее ссылается.

Терминология

argc
argv
atexit
calloc
const
exit
EXIT_FAILURE
EXIT_SUCCESS
make
makefile
raise
realloc
signal
signal.h
stdarg.h
tmpfile
va_arg
va_end
va_list
va_start
volatile
аргументы командной строки
библиотека обработки сигналов
внешняя компоновка
внутренняя компоновка
временный файл
динамические массивы
исключительная ситуация с плавающей точкой
конвейер

нарушение сегментации
недопустимая инструкция
переадресация ввода/вывода
перехват
прерывание
символ переадресации ввода <
символ переадресации вывода >
символ присоединения вывода >>
событие
спецификатор класса памяти **extern**
спецификатор класса памяти **static**
список аргументов переменной длины
суффикс **float** (**f** или **F**)
суффикс **long double** (**l** или **L**)
суффикс **long int** (**l** или **L**)
суффикс **unsigned int** (**u** или **U**)
суффикс **unsigned long** (**ul** или **UL**)

Распространенные ошибки программирования

14.1. Многоточие в середине списка параметров функции. Многоточие может находиться только в конце списка параметров.

Советы по переносимости программ

14.1. Некоторые операционные системы не поддерживают имена глобальных переменных или функций длиной более шести символов. Об этом не следует забывать при разработке программ, предназначенных для использования на различных машинах.

Советы по повышению эффективности

14.1. Глобальные переменные увеличивают производительность, потому что они доступны напрямую из любой функции — дополнительные затраты на передачу данных функциям таким образом устраняются.

14.2. Рассмотрите возможность использования двоичных файлов вместо текстовых в приложениях, требующих высокой производительности.

14.3. Оператор **goto** может использоваться для эффективного выхода из глубоких уровней вложенных управляющих структур.

Общие методические замечания

14.1. Следует избегать применения глобальных переменных за исключением случаев, когда быстродействие приложения является критическим фактором, поскольку они нарушают принцип минимума привилегий.

14.2. Создание программ из нескольких исходных файлов способствует повторному использованию кода и систематизированному конструированию программного обеспечения. Функции могут использоваться во многих приложениях. В таких случаях следует сохранять эти функции в отдельных исходных файлах, и каждый исходный файл должен иметь соответствующий ему заголовок, содержащий прототипы функций. Это позволяет программистам использовать в различных программах один и тот же код, включая в них соответствующий заголовочный файл и компилируя эти программы вместе с уже существующим исходным файлом.

14.3. Оператор **goto** должен использоваться только в приложениях, ориентированных на высокую производительность. Оператор **goto** не является структурным и программы с этим оператором, как правило, труднее отлаживать, сопровождать и модифицировать.

Упражнения для самоконтроля

14.1. Заполните пропуски в каждом из следующих утверждений.

a) Символ _____ используется для переключения ввода данных с клавиатуры на считывание из файла.

b) Символ _____ используется для переключения вывода данных, при котором они вместо вывода на экран помещаются в файл.

c) Символ _____ используется для присоединения вывода программы в конец файла.

d) _____ используется для того, чтобы направить выходной поток одной программы во входной поток другой.

e) _____ в списке параметров функции указывает, что функция может принимать переменное количество аргументов.

f) Макрос _____ должен быть вызван перед обращением к аргументам списка переменной длины.

g) Макрос _____ используется для доступа к отдельным аргументам в списке аргументов переменной длины.

h) Макрос _____ упрощает нормальный возврат из функции, на список аргументов которой ссылался макрос **va_start**.

i) Аргумент _____ функции **main** получает количество аргументов в командной строке.

j) Аргумент _____ функции **main** сохраняет аргументы командной строки как символьные строки.

k) Утилита UNIX _____ считывает файл _____, который содержит инструкции для компиляции и компоновки программ, состоящих из нескольких исходных файлов. Утилита перекомпилирует файл лишь в том случае, если он был изменен со времени последней компиляции.

l) Функция _____ принудительно завершает выполнение программы.

m) Функция _____ регистрирует функцию, которую необходимо вызвать при успешном завершении программы.

n) Модификатор типа _____ указывает, что объект не должен изменяться после инициализации.

o) _____ целого типа или типа с плавающей точкой может быть добавлен к целой константе или константе с плавающей точкой для определения точного типа константы.

p) Функция _____ открывает временный файл, который существует до тех пор, пока не будет закрыт или пока программа не завершится.

q) Функция _____ может использоваться для перехвата непредвиденных событий.

r) Функция _____ программно генерирует сигнал.

s) Функция _____ динамически выделяет память для массива и присваивает элементам нулевые значения.

t) Функция _____ изменяет размер динамически выделенного ранее блока памяти.

Ответы на упражнения для самоконтроля

14.1. a) переадресации ввода (<), b) переадресации вывода (>), c) присоединения вывода (>>), d) Конвейер (|), e) Многоточие (...), f) **va_start**, g) **va_arg**, h) **va_end**, i) **argc**, j) **argv**, k) **make**, **makefile**, l) **exit**, m) **atexit**, n) **const**, o) Суффикс, p) **tmpfile**, q) **signal**, r) **raise**, s) **calloc**, t) **realloc**.

Упражнения

14.2. Напишите программу для вычисления последовательности целых чисел, которые передаются функции **product** со списком аргументов переменной длины. Проверьте вашу функцию, вызвав ее несколько раз, каждый раз с новым количеством аргументов.

14.3. Напишите программу, которая выводит на печать аргументы командной строки программы.

14.4. Напишите программу, упорядочивающую массив целых чисел в восходящем или нисходящем порядке. Программа должна воспринимать аргументы командной строки — либо ключ **-a** для восходящего порядка, либо **-d** для нисходящего порядка. (Примечание: это стандартный формат для передачи программе опций в системе UNIX.)

14.5. Напишите программу, вставляющую пробелы между всеми символами в файле. Программа должна сначала записать содержимое обрабатываемого файла во временный файл, с пробелами между символами, после чего скопировать файл обратно в исходный файл. При этой операции первоначальное содержимое файла должно быть переписано.

14.6. Изучите руководство по вашей системе, чтобы определить, какие сигналы поддерживаются библиотекой обработки сигналов (**signal.h**). Напишите программу, которая обрабатывает стандартные сигналы **SIGABRT** и **SIGINT**. Программа должна тестировать перехват этих сигналов посредством вызова функции **abort** для генерации сигнала типа **SIGABRT** и ввода **<ctrl>c** для сигнала типа **SIGINT**.

14.7. Напишите программу, которая динамически выделяет массив целых чисел. Размер массива должен вводиться с клавиатуры. Элементам массива присваиваются значения, введенные с клавиатуры. Выведите значения массива на печать. После этого выполните повторное выделение памяти под массив размером в 1/2 от текущего

количества элементов. Выведите на печать значения, оставшиеся в массиве, чтобы убедиться, что они совпадают с первой половиной исходного массива.

14.8. Напишите программу, которая принимает из командной строки два аргумента, являющихся именами файлов, считывает посимвольно первый файл и записывает символы во второй файл в обратном порядке.

14.9. Напишите программу, использующую оператор **goto** для моделирования вложенной структуры циклов, которая выводит на печать показанный ниже квадрат, составленный из символов звездочки:

```
*****
*   *
*   *
*   *
*****
```

Программа должна выполнять только следующие операторы **printf**:

```
printf("*");
printf(" ");
pritf("\n");
```

глава

15

C++ как «улучшенный» C



Цели

- Познакомиться с усовершенствованиями языка C, реализованными в C++.
- Оценить важность C как основы для дальнейшего изучения программирования, в частности языка C++.

Содержание

15.1. Введение
15.2. Однострочные комментарии C++
15.3. Потоковый ввод/вывод C++
15.4. Объявления в C++
15.5. Создание новых типов данных в C++
15.6. Прототипы функций и контроль соответствия типов
15.7. Встроенные функции
15.8. Параметры-ссылки
15.9. Модификатор const
15.10. Динамическое распределение памяти с помощью new и delete
15.11. Параметры, используемые по умолчанию
15.12. Унарная операция разрешения области действия
15.13. Перегрузка функций
15.14. Спецификации внешней связи
15.15. Шаблоны функций

Резюме • Распространенные ошибки программирования • Хороший стиль программирования • Советы по переносимости программ • Советы по повышению эффективности • Общие методические замечания • Упражнения для самоконтроля • Ответы к упражнениям для самоконтроля • Упражнения • Рекомендуемая литература • Приложение: ресурсы C++

15.1. Введение

Начиная с этого момента мы будем заниматься языком C++. C++ улучшает многие характеристики языка C и включает в себя средства *объектно-ориентированного программирования* (OOP), обещающие значительно повысить производительность труда программистов и качество программного обеспечения, равно как степень его повторного использования. В этой главе обсуждаются некоторые из «улучшений», которые C++ вводит в старый C.

Проектировщики С и разработчики его первых версий и не предполагали, что этот язык станет своего рода феноменом (то же самое относится и к операционной системе UNIX). Когда язык программирования укореняется так глубоко, как это произошло с С, появление новых потребностей диктует развитие языка, а не простую замену его новым языком.

Язык C++ был разработан Бьерном Страуструпом в Bell Laboratories (St86) и первоначально назывался «С с классами». Название C++ включает в себя операцию инкремента языка С (++) и указывает на то, что C++ — версия С с расширенными возможностями.

C++ является «надмножеством» С, поэтому программисты могут использовать компилятор C++ для компиляции существующих программ на С и затем постепенно переводить эти программы на C++. Некоторые ведущие продавцы программного обеспечения уже поставляют компиляторы C++, не предлагая при этом отдельных средств разработки для языка С.

Пока не существует стандарта ANSI для C++, хотя комитет ANSI разрабатывает свои предложения. Многие полагают, что наиболее важные среды программирования на С будут переведены на C++ к середине 1990-х годов[1].

15.2. Однострочные комментарии C++

Программисты часто вставляют небольшой комментарий в конце строки кода. Язык С требует, чтобы комментарий был заключен между символами /* и */. C++ позволяет начинать комментарий с // и использовать остаток строки для текста комментария; символ конца строки автоматически завершает комментарий. Эта форма комментария экономит несколько нажатий клавиш, однако она применима только для комментариев, не продолжающихся на следующую строку.

Например, комментарий С

```
/* Это однострочный комментарий. */
```

требует обоих разделителей /* и */ даже для однострочного комментария. Однострочная версия языка C++:

```
// Это однострочный комментарий.
```

Для многострочных комментариев, например

```
/* Это один из способов       */
/* написания понятных много-  */
/* строчных комментариев.     */
```

нотация C++ может показаться более краткой:

```
// Это один из способов
// написания понятных много-
// строчных комментариев.
```

Однако не забывайте, что комментарии в языке С могут охватывать несколько строк, так что на самом деле требуется только одна пара ограничителей комментария (а не три пары, использованных выше):

```
/* Это один из способов
   написания понятных много-
   строчных комментариев. */
```

---

1 В настоящее время стандарт ANSI/ISO C++ уже сформулирован и одобрен. Это, однако, не исключает дальнейшего развития и модификации языка. — *Прим. ред.*

Поскольку C++ является надмножеством C, в программе на C++ приемлемы обе формы комментария.

Распространенная ошибка программирования 15.1

Забывают закрыть комментарий в стиле C символами */.

Хороший стиль программирования 15.1

Использование комментариев в стиле языка C++ с // помогает избегать ошибок незавершенного комментария, которые возникают при пропуске символов */, закрывающих комментарии C.

Это на вид простое упущение может приводить к глубинным трудноуловимым ошибкам, некоторые из которых могут проскользнуть мимо компилятора необнаруженными. Результатом пропуска */ является то, что компилятор предполагает, что комментарий еще не закончен, следовательно, он продолжается во всех последующих строках до тех пор, пока не встретится другой символ комментария */ или будет достигнут конец исходного файла. Это может привести к игнорированию компилятором ключевого фрагмента программы.

15.3. Потоковый ввод/вывод C++

Язык C++ предусматривает альтернативную обращениям к функциям **printf** и **scanf** возможность обработки ввода/вывода стандартных типов данных и строк. Например, простой диалог

```
printf("Enter new tag: ");
scanf("%d", &tag);
printf("The new tag is: %d\n", tag);
```

записывается на C++ в виде

```
cout << "Enter new tag: ";
cin >> tag;
cout << "The new tag is: " << tag << '\n';
```

В первом операторе используется *стандартный выходной поток* **cout** и операция << (*операция передачи в поток*, произносится «послать в»). Оператор читается так:

*"**Enter new tag:** " послать строку в выходной поток **cout**.*

Обратите внимание, что операция передачи в поток обозначается так же, как поразрядная операция сдвига влево. Во втором операторе используется стандартный входной поток **cin** и операция >> (*извлечения из потока*, произносится «взять из»). Этот оператор читается

*Взять из входного потока значение для переменной **tag**.*

Обратите внимание, что операция извлечения из потока становится операцией поразрядного сдвига вправо, если левый параметр является целочисленным типом. Операции передачи и извлечения из потока, в отличие от функций **printf** и **scanf**, не требуют форматирующих строк и спецификаторов преобразования для указания на тип входных и выходных данных. В C++ есть много примеров, подобных этому, когда он автоматически «знает», какие типы должны участвовать в операциях. Также обратите внимание на то, что при испо-

льзовании с операцией извлечения из потока переменной **tag** не предшествует операция взятия адреса &, которую требует **scanf**.

Для организации потокового ввода/вывода программы на C++ должны включать заголовочный файл **iostream.h**. Программа на рис. 15.1 просит ввести переменные **myAge** и **friendsAge** и определяет, старше ли вы вашего друга, моложе его или ваш возраст совпадает. Обратите внимание на то, что переменные **myAge** и **friendsAge** объявляются непосредственно перед ссылкой на них в операторах ввода. В главе 21, «Потоковый ввод/вывод», дается детальное описание особенностей потокового ввода/вывода языка C++.

```
// Пример простого потокового ввода/вывода
#include <iostream.h>

main ()
{
   cout << "Enter your age: ";
   int myAge;
   cin >> myAge;

   cout << "Enter your friend's age: ";
   int friendsAge;
   cin >> friendsAge;

   if (myAge > friendsAge)
      cout << "You are older.\n";
   else
      if (myAge < friendsAge)
         cout << "You are younger.\n";
      else
         cout << "You and your friend are the same age.\n";

   return 0;
}
```

```
Enter your age: 23
Enter your friend's age: 20
You are older.

Enter your age: 20
Enter your friend's age: 23
You are younger.

Enter your age: 20
Enter your friend's age: 20
You and your friend are the same age.
```

Рис. 15.1. Потоковый ввод/вывод с операциями передачи и извлечения из потока

Хороший стиль программирования 15.2

Применение ориентированного на потоки ввода/вывода в стиле языка C++ делает программы более удобочитаемыми (и менее уязвимыми для ошибок), чем их эквиваленты на C с вызовами функций **printf** и **scanf**.

15.4. Объявления в C++

В языке C все объявления должны находиться в начале блока до любых исполняемых операторов. В C++ объявления могут размещаться всюду, где может стоять исполняемый оператор, при условии, что они предшествуют использованию того, что объявляется. Например, во фрагменте кода

```
cout << "Enter two integers: ";
int x, y;
cin >> x >> y;
cout << "The sum of " << x << " and " << y
     << " is " << x + y << '\n';
```

переменные **x** и **y** объявляются после исполняемого оператора **cout**, но до их использования в последующем операторе **cin**. Кроме того, переменные могут объявляться в разделе инициализации структуры **for** — такие переменные остаются в области действия до конца блока, в котором определена структура **for**[1]. Например,

```
for (int i = 0; i <= 5; i++)
   cout << i << '\n';
```

объявляет целую переменную **i** и инициализирует ее значением **0** в структуре **for**.

Область действия локальной переменной в языке C++ начинается с ее объявления и распространяется до закрывающей правой фигурной скобки (}). Следовательно, операторы, предшествующие объявлению переменной, не могут ссылаться на эту переменную, даже если эти операторы находятся в том же самом блоке. Объявления переменных не могут помещаться в условиях структур **while**, **do/while**, **for** или **if**.

Хороший стиль программирования 15.3

Размещение объявлений переменных поблизости от их первого использования может сделать программу более ясной.

Распространенная ошибка программирования 15.2

Объявление переменной после ссылки на нее в каком-либо операторе.

15.5. Создание новых типов данных в C++

Язык C++ предусматривает возможность определения пользователем новых типов с помощью ключевых слов **enum**, **struct**, **union** и нового ключевого слова **class**. Как и в языке C, перечисления в C++ объявляются при помощи **enum**. Однако в отличие от C перечисление в C++ после его объявления становится новым типом. Для объявления переменных этого нового типа ключевое слово **enum** не требуется. То же самое верно для структур, объединений и классов. Например, в объявлениях

---

1 В последних редакциях стандарта C++ постулируется, что область действия переменных, объявленных в заголовке структуры **for**, *ограничивается телом этой структуры*. Вне тела оператора **for** такие переменные недоступны. — *Прим. ред.*

```
enum Boolean {FALSE, TRUE};

struct Name {
   char first[10];
   char last[10];
}

union Number {
   int i;
   float f;
};
```

создаются три новых типа данных с именами-этикетками **Boolean**, **Name** и **Number**. Эти имена-этикетки могут быть использованы для объявления переменных следующим образом:

```
Boolean done = FALSE;
Name student;
Number x;
```

Эти объявления создают переменную **done** типа **Boolean** (инициализированную значением **FALSE**), переменную **student** типа **Name** и переменную **x** типа **Number**.

Как и в языке C, перечислениям присваиваются значения, начиная по умолчанию с нуля, и каждый последующий элемент увеличивается на единицу. Таким образом, в перечислении **Boolean FALSE** присваивается значение **0** и **TRUE** значение **1**. Любому элементу в перечислении может быть присвоено целочисленное значение. Последующие элементы, которым не присвоено определенного значения, автоматически получают увеличенное на единицу значение предыдущего элемента.

15.6. Прототипы функций и контроль соответствия типов

Прототипы функций позволяют компилятору C контролировать правильность вызова функций с точки зрения соответствия типов. В ANSI C прототипы функций являются необязательными. В C++ для всех функций требуются их прототипы. Функция, определенная в файле до первого обращения к ней, не требует отдельного прототипа. В этом случае заголовок функции действует как ее прототип. C++, как и ANSI C, требует объявления в круглых скобках всех параметров функции в ее определении и прототипе. Например, функция **square**, которая принимает целочисленный параметр и возвращает целое значение, имеет прототип

```
int square (int);
```

Функции, которые не возвращают значения, объявляются с типом возвращаемого значения **void**.

Распространенная ошибка программирования 15.3

Попытка возвратить значение из функции типа **void** или использовать результат вызова функции **void**.

В языке C для определения пустого списка параметров в круглые скобки помещается ключевое слово **void**. Если в круглых скобках прототипа функции на C ничего не содержится, то для этой функции полностью отключается проверка параметров и не делается никаких предположений относительно числа параметров функции и их типа. При обращениях к этой функции могут передаваться любые параметры без сообщения компилятора о каких бы то ни было ошибках.

В C++ при задании пустого списка параметров в круглых скобках либо записывается слово **void**, либо вообще ничего не записывается. Объявление

```
void print();
```

сообщает, что функция **print** не принимает параметров и не возвращает значения. На рис. 15.2 показаны оба принятых в C++ способа объявления функций, которые не принимают параметров.

```
// Пример функции, не принимающей параметров
#include <iostream.h>

void f1();
void f2(void);

main()
{
   f1();
   f2();

   return 0;
}

void f1()
{
   cout << "Function f1 takes no argumens\n";
}

void f2(void)
{
   cout << "Function f2 also takes no arguments\n";
}
```

```
Function f1 takes no argumens
Function f2 also takes no arguments
```

Рис. 15.2. Два способа объявления функций, не принимающих параметров

Совет по переносимости программ 15.1

Смысл пустого списка параметров функции в языке C++ разительно отличается от его смысла в C. В C это означает, что блокируется любая проверка параметров. В C++ это означает, что функция не принимает параметров. Поэтому программы на C, использующие такие объявления, могут выполняться иначе при компиляции в C++.

Распространенная ошибка программирования 15.4

Программы на C++ не компилируются, если для каждой функции не объявлен ее прототип или она не определена до обращения к ней.

15.7. Встроенные функции

Реализация программы в виде набора функций хороша с точки зрения разработки программного обеспечения, но обращения к функциям связаны с определенными накладными расходами времени исполнения. В C++ предусмотрены *встроенные функции*, позволяющие уменьшить накладные расходы при вызове функций — особенно для функций небольшого размера. Модификатор **inline**, помещенный перед типом возвращаемого функцией значения в определении функции, рекомендует компилятору генерировать в месте вызова функции копию ее кода (если это возможно) с тем, чтобы избежать вызова функции. Компромисс состоит в том, что вместо одной копии функции, в которую передается управление всякий раз при ее вызове, в программу вставляются многие копии кода этой функции. Компилятор может игнорировать **inline** и обычно так и поступает, за исключением самых маленьких функций.

Общее методическое замечание 15.1

Любое изменение встроенной функции требует перекомпиляции всех функций, являющихся ее клиентами. Это может быть существенным в определенных ситуациях, связанных с разработкой и сопровождением программ.

Встроенные функции обладают преимуществами перед макросами препроцессора (см. главу 13), также расширяющимися в месте «вызова» макроса. Одним из преимуществ является то, что встроенные функции практически не отличаются от любых других функций языка C++. Следовательно, при обращениях к встроенным функциям выполняется надлежащий контроль соответствия типов; макросы препроцессора не поддерживают контроля соответствия типов. Другим преимуществом является то, что встроенные функции исключают непредвиденные побочные эффекты, связанные с неправильным использованием макросов. И наконец, встроенные функции можно отлаживать с помощью отладчика. Отладчик не распознает макросы препроцессора в качестве особых компонентов программы, поскольку они представляют собой просто текстовые замены, выполняемые препроцессором перед компиляцией. Отладчик может помочь найти логические ошибки, являющиеся результатом макроподстановок, но не может отнести эти ошибки к определенным макроопределениям.

Хороший стиль программирования 15.4

Модификатор **inline** следует использовать только с небольшими, часто используемыми функциями.

Совет по повышению эффективности 15.1

Использование встроенных функций может привести к уменьшению времени выполнения программы, но при этом увеличить ее размер.

В программе на рис. 15.3 используется встроенная функция **cube** для расчета объема куба со стороной **s**. Ключевое слово **const** в списке параметров функции **cube** сообщает компилятору, что функция не изменяет переменную **s**. Ключевое слово **const** более подробно обсуждается в разделе 15.9.

```
// Использование встроенной функции для расчета
// объема куба
#include <iostream.h>

inline float cube(const float s)  { return s * s * s; }

main ()
{
   cout << "Enter the side lenght of your cube: ";

   float side;

   cin >> side;
   cout << "Volume of cube with side "
        << side << " is " << cube(side) << '\n' ;

   return 0;
}
```

```
Enter the side lenght of your cube: 3.5
Volume of cube with side 3.5 is 42.875
```

Рис. 15.3. Использование встроенной функции для вычисления объема куба

Макросы препроцессора представляют собой операции, определяемые в директиве препроцессора **#define**. Макрос состоит из имени (идентификатора макроса) и текста замены. Макросы могут определяться со списком параметров или без него. Макросы без параметров обрабатываются подобно символическим константам — текст замены подставляется в программе вместо идентификатора макроса. Макросы с параметрами подставляют параметры в заменяющий текст, и затем происходит расширение макроопределения в программе.

Рассмотрим следующее макроопределение с одним параметром для вычисления площади квадрата:

```
#define VALIDSQUARE(x)     (x) * (x)
```

Препроцессор находит в программе каждое вхождение **VALIDSQUARE(x)**, подставляет **x** (значение или выражение) в текст замены и расширяет макрос в программе. Например,

```
cout << VALIDSQUARE(7);
```

расширяется до

```
cout << (7) * (7);
```

Круглые скобки в тексте замены обеспечивают правильный порядок оценки выражения макроса. Например,

```
cout << VALIDSQUARE(2 + 3);
```

раскрывается в

```
cout << (2 + 3) * (2 + 3);
```

что правильно оценивается значением **5 * 5 = 25**. Препроцессор не оценивает выражений, задаваемых в качестве параметров макроса; он просто заменяет каждое вхождение имени параметра в тексте замены копией всего выражения. Следовательно, для того, чтобы гарантировать правильную оценку параметров, необходимы круглые скобки.

Рассмотрим соответствующие макроопределение без круглых скобок в тексте замены:

```
#define VALIDSQUARE(x)     x * x
```

Если мы укажем в качестве параметра выражение **2 + 3**, макрос раскроется следующим образом:

```
2 + 3 * 2 + 3
```

Это выражение неправильно оценивается значением **2 + 6 + 3 = 11**, поскольку умножение имеет более высокий приоритет, чем сложение.

Для устранения проблем, связанных с макросами, устранения непроизводительных затрат, связанных с обращениями к функциям, и обеспечения проверки типа параметров в C++ предусмотрены встроенные функции. Параметры встроенных функций оцениваются до их «передачи» в функцию. Следовательно, нет необходимости заключать в круглые скобки каждое вхождение каждого параметра в ее определении.

Встроенная функция

```
inline int square(int x) { return x * x; }
```

вычисляет квадрат своего целочисленного параметра **x**. При вызове **square(2 + 3)** происходит оценка параметра **2 + 3 = 5** и его значение подставляется в тело функции. В программе на рис. 15.4 показаны макросы **VALIDSQUARE** и **INVALIDSQUARE** и встроенная функция **square**.

```
// Примеры корректных и некорректных макросов
// и встроенных функций
#include <iostream.h>

#define VALIDSQUARE(x)     (x) * (x)
#define INVALIDSQUARE(x)   x * x

inline int square(int x)   { return x * x; }

main()
{
   cout << " VALIDSQUARE(2 + 3) = "
        << VALIDSQUARE(2 + 3)
        << "\nINVALIDSQUARE(2 + 3) = "
        << INVALIDSQUARE(2 + 3)
        << "\n       square(2 + 3) = "
        << square(2 + 3) << '\n' ;

   return 0;
}
```

```
 VALIDSQUARE(2 + 3) = 25
INVALIDSQUARE(2 + 3) = 11
       square(2 + 3) = 25
```

Рис. 15.4. Макросы препроцессора и встроенные функции

На рис. 15.5 приведен полный перечень ключевых слов языка C++. На рисунке показаны ключевые слова, общие для С и C++, и затем отдельно ключевые слова, используемые только в C++. Далее в книге подробно разъясняется каждое из новых ключевых слов языка C++.

Ключевые слова языка C++				
Языки C и C++				
auto	break	case	char	const
continue	default	do	double	else
enum	extern	float	for	goto
if	int	long	register	return
short	signed	sizeof	static	struct
switch	typedef	union	unsigned	void
volatile	while			
Только язык C++				
asm	Определяемое реализацией средство использования кода ассемблера в программах C++ (см. руководства по вашей системе).			
catch	Обрабатывает исключение, генерируемое оператором **throw**.			
class	Определяет новый класс. Могут быть созданы объекты этого класса.			
delete	*Разрушает динамический объект в памяти, созданный операцией* **new**.			
friend	Объявляет функцию или класс в качестве дружественного другому классу. Друзья могут обращаться ко всем элементам данных и элементам-функциям класса.			
inline	Сообщает компилятору, что для данной функции вместо вызова должен генерироваться встроенный код.			
new	Динамически распределяет память под объект в «области свободной памяти» — дополнительной памяти, доступной программе во время выполнения. Автоматически определяет размер объекта.			
operator	Объявляет перегруженную операцию.			
private	Закрытый элемент класса, доступный для элементов-функций и функций-друзей класса.			
protected	Расширенная форма доступа private; к защищенным элементам могут также обращаться элементы-функции производных классов.			
public	Элемент класса, доступный для любой функции.			
template	Описывает шаблон класса или функции (параметризованный тип).			
this	Указатель, неявно объявляемый в каждой не статической функции-элементе класса. Он указывает на объект, для которого была вызвана эта функция.			
throw	Передает управление обработчику исключительных ситуаций или завершает выполнение программы, если соответствующий обработчик не может быть найден.			
try	Создает блок, содержащий набор операторов, которые могут генерировать исключительные ситуации, и делает возможной обработку всех генерируемых исключений.			
virtual	Объявляет виртуальную функцию.			

Рис. 15.5. Ключевые слова языка C++

Распространенная ошибка программирования 15.5

C++ является развивающимся языком, и некоторые его возможности могут не поддерживаться вашим компилятором. Использование нереализованных свойств языка приводит к синтаксическим ошибкам.

15.8. Параметры-ссылки

В языке С при всех обращениях к функциям параметры передаются по значению. Передача параметра по ссылке имитируется в С путем передачи в функцию указателя на объект и последующего доступа к этому объекту путем разыменования указателя в вызываемой функции. Не забывайте, что имена массивов в С уже являются (константными) указателями, поэтому массивы автоматически передаются путем имитации передачи параметра по ссылке. Другие языки программирования предлагают явные формы передачи параметра по ссылке — как, например, параметры **var** в Паскале. C++ устраняет этот недостаток языка С, позволяя объявлять *параметры-ссылки*.

Параметр-ссылка является псевдонимом для соответствующего ему параметра. Чтобы указать, что параметр функции передается по ссылке, просто поместите после типа параметра в прототипе функции символ амперсанда (&) (точно так же, как вы поместили бы за типом параметра символ * для указания на то, что параметр является указателем). То же самое относится и к спецификации этого параметра в заголовке функции. Например, объявление

```
int &count
```

в заголовке функции можно прочитать как «переменная **count** является ссылкой на **int**». При обращении к функции просто укажите для параметра имя переменной, и она будет автоматически передана по ссылке. После этого ссылка на локальное имя параметра в теле функции на самом деле относится к исходной переменной в вызывающей функции, и исходная переменная может непосредственно модифицироваться вызываемой функцией.

На рис. 15.6 сравниваются передача по значению, передача по ссылке с использованием указателей и передача по ссылке с использованием параметров-ссылок. Параметры в вызовах функций **squareByValue** и **squareByReference** идентичны. Нельзя сказать, изменяет ли каждая из этих двух функций свои параметры, без рассмотрения прототипов этих функций или их определений. По этой причине некоторые программисты предпочитают передавать в функции модифицируемые параметры посредством указателей, а немодифицируемые параметры в виде ссылок на константы. Ссылки на константы обеспечивают эффективность, достигаемую при передаче указателей, и предотвращают изменение параметров вызывающей программы. Чтобы определить ссылку на константу, поместите модификатор **const** перед спецификатором типа в объявлении параметра (в разделе 15.9 модификатор **const** обсуждается более подробно). Обратите внимание на положение символов * и & в списках параметров этих двух функций. Некоторые программисты предпочитают писать **int* bPtr**, а не **int *bPtr** и **int& cRef**, а не **int &cRef**.

Хороший стиль программирования 15.5

Используйте указатели для передачи параметров, которые могут быть изменены вызываемой функцией, и ссылки на константы для передачи параметров большого размера, не подлежащих изменению.

```
// Сравнение передачи параметра по значению, передачи параметра
// по ссылке с использованием указателей и передачи параметра
// по ссылке с использованием ссылок
#include <iostream.h>

int squareByValue(int);
void squareByPointer(int *);
void squareByReference(int &);

main()
{
   int x = 2, y = 3, z = 4;

   cout << "x = " << x << " before squareByValue\n"
        << "Value returned by squareByValue: "
        << squareByValue(x)
        << "\nx = " << x << " after squareByValue\n\n " ;

   cout << "y = " << y << " before squareByPointer\n " ;
   squareByPointer(&y) ;
   cout << "y = " << y << " after squareByPointer\n\n" ;

   cout << "z = " << z << " before squareByReference\n" ;
   squareByReference(z) ;
   cout << "z = " << z << " after squareByReference\n" ;

   return 0 ;
}

int squareByValue(int a)
{
   return a *= a ; // параметр в вызывающей функции не изменяется
}

void squareByPointer(int *bPtr)
{
   *bPtr *= *bPtr ; // параметр в вызывающей функции изменяется
}

void squareByReference(int &cRef)
{
   cRef *= cRef ;    // параметр в вызывающей функции изменяется
}
```

```
x = 2  before squareByValue
Value returned by squareByValue: 4
x = 2  after squareByValue

y = 3  before squareByPointer
y = 9  after squareByPointer

z = 4  before squareByReference
z = 16  after squareByReference
```

Рис. 15.6. Пример передачи параметра по ссылке

Распространенная ошибка программирования 15.6

Поскольку доступ к параметрам-ссылкам в теле вызываемой функции производится просто по имени, программист может по неосторожности обращаться с параметрами-ссылками как с параметрами, передаваемыми по значению. Это может привести к неожиданным побочным эффектам, если исходные копии переменных будут изменены вызывающей функцией.

Совет по повышению эффективности 15.2

Для больших объектов используйте параметр-ссылку с **const**, чтобы имитировать внешний вид и надежность передачи параметра по значению, но избежать накладных расходов, связанных с передачей копии большого объекта.

Ссылки также могут служить в качестве псевдонимов для других переменных внутри некоторой функции. Например, код

```
int count = 1;     // объявляет целочисленную переменную count
int &c = count;    // создает с в качестве псевдонима для count
++c;               // увеличивает значение переменной count
                   // (используя ее псевдоним)
```

увеличивает значение переменной **count**, используя ее псевдоним **c**. Переменные-ссылки должны инициализироваться при их объявлении (см. рис. 15.7 и рис. 15.8) и не могут быть переназначены в качестве псевдонимов других переменных. Как только ссылка объявлена в качестве псевдонима другой переменной, все операции, выполняемые якобы над псевдонимом (т.е. ссылкой), на самом деле выполняются непосредственно над самой исходной переменной. Псевдоним представляет собой просто другое имя для исходной переменной — для него не резервируется память.

Ссылки не могут разыменовываться путем применения операции косвенной адресации (*). Над ссылками нельзя выполнять действия «ссылочной арифметики» (как это делается в арифметике указателей, чтобы обращаться к различным элементам массива). В число других недопустимых операций со ссылками входят указание на ссылки, взятие адресов ссылок и сравнение ссылок (на самом деле каждая из этих операций не вызывает синтаксической ошибки; каждая операция выполняется над переменной, для которой ссылка является псевдонимом).

Функции могут возвращать указатели или ссылки, но это может представлять опасность. При возврате указателя или ссылки на переменную вызываемой функции она должна быть объявлена статической. Иначе указатель или ссылка будут относиться к переменной с автоматическим классом памяти, которая разрушается при завершении функции; говорят, что такая переменная является «неопределенной» и поведение программы может стать непредсказуемым.

Распространенная ошибка программирования 15.7

Отсутствие инициализации переменной-ссылки при ее объявлении.

Распространенная ошибка программирования 15.8

Попытка переназначить ранее объявленную ссылку в качестве псевдонима другой переменной.

```
// Ссылки должны быть инициализированы
#include <iostream.h>

main()
{
   int x = 3 , &y;     // Ошибка: y должна быть инициализирована

   cout << "x = " << x << '\n'
        << "y = " << y << '\n' ;
   y = 7 ;
   cout << "x = "" << x << '\n'
        << "y = " << y << '\n' ;

   return 0 ;
}
```

```
Compiling FIG15_7.CPP:
Error FIG15_7.CPP 6: Reference variable 'y' must be initialized
```

Рис. 15.7. *Попытка использования неинициализированной ссылки*

```
// Ссылки должны быть инициализированы
#include <iostream.h>

main ()
{
   int x = 3, &y = x ;  // y теперь является псевдонимом для x

   cout << "x = " << x << '\n'
        << "y = " << y << '\n' ;
   y = 7 ;
   cout << "x = " << x << '\n'
        << "y = " << y << '\n' ;

   return 0 ;
}
```

```
x = 3
y = 3
x = 7
y = 7
```

Рис. 15.8. Использование инициализированной ссылки

Распространенная ошибка программирования 15.9

Попытка разыменования переменной-ссылки с помощью операции косвенной адресации (не забывайте, что ссылка является псевдонимом для переменной, а не указателем на нее).

Распространенная ошибка программирования 15.10

Возвращение указателя или ссылки на автоматическую переменную, определенную в вызываемой функции.

15.9. Модификатор const

В разделе 15.7 иллюстрировалось использование модификатора **const** в списке параметров функции для указания того, что передаваемый в функцию параметр не может быть в ней изменен. Модификатор **const** может также применяться для объявления так называемых «переменных-констант» (вместо объявления символических констант в препроцессоре с помощью **#define**), например

```
const float PI = 3.14159;
```

Это объявление создает «переменную-константу» **PI** и инициализирует ее значением **3.14159**. Переменная должна быть инициализирована константным выражением при ее объявлении и не может быть в дальнейшем изменена (рис. 15.9 и рис. 15.10). Переменные-константы называют также именованными константами или переменными только для чтения. Обратите внимание, что термин «переменная-константа» является оксюмороном, т.е. противоречивым выражением вроде «тщедушного великана» или «холодного ожога».

```
// Константный объект должен быть инициализирован
main ()
{
   const int x ;  // Ошибка: x должен быть инициализирован

   x = 7 ;        // Ошибка: не могу изменить константную переменную

   return 0 ;
}
```

```
Compiling FIG15_9.CPP:
Error FIG15_9.CPP 4: Constant variable 'x' must be initialized
Error FIG15_9.CPP 6: Cannot modify a const object
```

Рис. 15.9. Объект с модификатором const должен быть инициализирован

```
// Использование константной переменной,
// инициализированной надлежащим образом
#include <iostream.h>

main ()
{
   const int x = 7 ; // инициализированная константная переменная

   cout << "The value of constant variable x is: "
        << x << '\n' ;

   return 0 ;
}
```

```
The value of constant variable x is: 7
```

Рис. 15.10. Корректная инициализация и использование переменной-константы

Константные переменные могут использоваться всюду, где возможно вхождение константного выражения. Например, следующее объявление мас-

сива использует переменную с модификатором **const** для задания размера массива:

```
const int arraySize = 100;
int array[arraySize];
```

Константные переменные могут также помещаться в заголовочные файлы.

Распространенная ошибка программирования 15.11

Использование переменных с модификатором **const** в объявлениях массивов и помещение переменных с модификатором **const** в заголовочные файлы (включаемые в различные исходные файлы одной и той же программы) запрещено в C, но разрешено в C++.

Хороший стиль программирования 15.6

Преимуществом использования переменных с модификатором **const** вместо символических констант является то, что переменные с модификатором **const** видимы для символического отладчика; константы, определяемые с помощью **#define**, для него невидимы.

Существуют и другие общепринятые способы применения модификатора **const**. Например, константный указатель может быть объявлен следующим образом:

```
int *const iPtr = &integer;
```

Здесь объявлено, что **iPtr** является константным указателем на целое число. Значение, на которое указывает **iPtr**, может быть изменено, однако **iPtr** не может быть присвоено значение, указывающее на другую ячейку памяти.

Указатель на константный объект может быть объявлен следующим образом:

```
const int *iPtr = &integer;
```

Здесь объявлено, что **iPtr** является указателем на целочисленную константу. Значение, на которое указывает **iPtr**, не может быть изменено с использованием **iPtr**, однако **iPtr** может быть присвоено значение, указывающее на другую область памяти. Таким образом, **iPtr** является указателем, посредством которого значение может быть прочитано, но не может быть изменено (по сути дела, указателем «только для чтения»). Компилятор защищает данные, на которые ссылаются указатели только для чтения, тем, что он не допускает присваивания таких указателей указателям, не являющимся указателями только для чтения.

Распространенная ошибка программирования 15.12

Инициализация переменной const неконстантным выражением, например, переменной, не объявленной с модификатором const.

Распространенная ошибка программирования 15.13

Попытка изменить константную переменную.

Распространенная ошибка программирования 15.14

Попытка изменить константный указатель.

15.10. Динамическое распределение памяти с помощью new и delete

Операции C++ ***new*** и ***delete*** предназначены для управления динамическим распределением памяти в программах. В ANSI C динамическое распределение памяти обычно выполняется при помощи стандартных библиотечных функций **malloc** и **free**. Рассмотрим объявление

```
typeName *ptr;
```

где **typeName** является любым типом (например, **int**, **float**, **char** и т. д.). В ANSI C следующий оператор динамически выделяет память объекту **typeName**, возвращает указатель на него типа **void** и присваивает этот указатель **ptr**:

```
ptr = malloc(sizeof(typeName));
```

Динамическое выделение памяти в языке С требует обращения к функции **malloc** и явной ссылки на операцию **sizeof** (или явного указания необходимого числа байт). Объект размещается в специальной области памяти, доступной программе во время выполнения. Кроме того, в реализациях С, относящихся к периоду до появления стандарта ANSI, указатель, возвращаемый функцией **malloc**, должен быть преобразован к соответствующему типу посредством приведения (**typeName ***).

В C++ оператор

```
ptr = new typeName;
```

выделяет память для объекта типа **typeName** из области *свободной памяти* программы (термин языка C++ для дополнительной памяти, доступной программе во время выполнения). Операция **new** автоматически создает объект соответствующего размера и возвращает указатель правильного типа. Если операция **new** не может выделить память, возвращается нулевой указатель (в C++ для представления нулевого указателя используют значение **0**, а не **NULL**).

Для освобождения памяти, выделенной для этого объекта, в C++ применяется следующий оператор:

```
delete ptr;
```

В языке С для освобождения памяти вызывается функция **free** с параметром **ptr**. Операция **delete** может быть использована только для освобождения памяти, выделенной операцией **new**. Применение операции **delete** к ранее освобожденному указателю может привести к непредвиденным ошибкам во время выполнения программы. Применение **delete** к нулевому указателю не оказывает никакого влияния на выполнение программы.

Распространенная ошибка программирования 15.15

Освобождение с помощью **delete** памяти, которая не была выделена операцией **new**.

C++ допускает применение *инициализатора* для вновь создаваемого объекта. Например,

```
float *thingPtr = new float (3.14159);
```

инициализирует создаваемый объект типа **float** значением **3.14159**.

Массивы также могут создаваться динамически с помощью операции **new**. Следующий оператор динамически выделяет память для одномерного массива из 100 целых чисел и присваивает указатель, возвращаемый операцией **new**, указателю на целое **arrayPtr**:

```
int *arrayPtr;
arrayPtr = new int[100];  // динамически создает массив
```

Для освобождения памяти, динамически выделенной для **arrayPtr** операцией **new[]**, используйте оператор

```
delete [] arrayPtr;
```

В главе 16 мы обсудим динамическое распределение памяти для объектов классов. Мы увидим, что операции **new** и **delete** выполняют и другие задачи (например, автоматически вызывают конструкторы и деструктор класса), что делает использование операций **new** и **delete** более мощным и более надежным, чем вызов функций **malloc** и **free**. В дальнейшем не забывайте о квадратных скобках в операции **delete**, освобождающей массив объектов.

Распространенная ошибка программирования 15.16

Применение операции **delete** без скобок ([и]) при удалении динамически созданных массивов (это существенно только для массивов, содержащих объекты пользовательских типов).

15.11. Параметры, используемые по умолчанию

Часто при обращении к функциям может передаваться особое значение аргумента. Программист может указать, что такого рода аргумент является *аргументом по умолчанию*, и задать для него соответствующее значение. Если такой аргумент при вызове функции опускается, в функцию автоматически передается значение по умолчанию.

Аргументы по умолчанию должны быть самыми правыми (последними) аргументами в списке параметров функции. При вызове функции с двумя или более аргументами по умолчанию, если опущенный аргумент не является в списке параметров крайним справа, то все аргументы справа от него также должны быть опущены. Аргументы по умолчанию должны быть заданы при первом появлении имени функции — обычно в ее прототипе в заголовочном файле. Аргументы по умолчанию также могут использоваться и со встроенными функциями.

На рис. 15.11 демонстрируется объявление аргументов по умолчанию при расчете объема прямоугольного параллелепипеда. Для всех трех аргументов были заданы значения по умолчанию, равные **1**. При первом обращении к встроенной функции **boxVolume** ни один аргумент не определен и, таким образом, используются все три значения по умолчанию. При втором обращении передается аргумент **length** и используются значения по умолчанию для аргументов **width** и **height**. При третьем обращении передаются аргументы **length** и **width** и значение по умолчанию используется только для аргумента **height**.

При последнем обращении передаются аргументы **length**, **width** и **height** и, таким образом, не используется ни одного значения по умолчанию.

```
// Использование аргументов по умолчанию
#include <iostream.h>

// Рассчитывает объем прямоугольного параллелепипеда
inline int boxVolume(int lenght = 1, int width = 1,
                     int height = 1 )
   { return length * width * height; }

main ()
{
   cout << "The default box volume is: "
        << boxVolume()
        << "\n\nThe volume of a box with length 10, \n"
        << "width 1 and height 1 is: "
        << boxVolume(10)
        << "\n\nThe volume of a box with length 10, \n"
        << "width 5 and height 1 is: "
        << boxVolume(10, 5)
        << "\n\nThe volume of a box with length 10, \n"
        << "width 5 and height 2 is: "
        << boxVolume(10, 5, 2)
        << '\n' ;

   return 0 ;
}
```

```
The default box volume is: 1

The volume of a box with length 10,
width 1 and height 1 is: 10

The volume of a box with length 10,
width 5 and height 1 is: 50

The volume of a box with length 10,
width 5 and height 2 is: 100
```

Рис. 15.11. Задание аргументов по умолчанию

Хороший стиль программирования 15.7

Задание аргументов по умолчанию может упростить отдельные вызовы функций. Однако некоторые программисты считают, что явное указание всех аргументов вызова делает программу более понятной.

Распространенная ошибка программирования 15.17

Определение и попытка использовать аргумент по умолчанию, который не является самым правым (последним) аргументом (если одновременно не являются аргументами по умолчанию все аргументы справа от него).

15.12. Унарная операция разрешения области действия

В C и C++ возможно объявление локальных и глобальных переменных с одним и тем же именем. В языке C, пока локальная переменная находится в области действия, все ссылки на имя этой переменной относятся к локальной переменной — глобальная переменная не видна в области действия локальной переменной. В языке C++ предусмотрена *унарная операция разрешения области действия* (::) для доступа к глобальной переменной, когда в области действия находится локальная переменная с тем же именем. Унарная операция разрешения области действия не может использоваться для доступа к переменной с этим же именем во внешнем блоке. К глобальной переменной можно обращаться непосредственно, без операции разрешения области действия, если имя глобальной переменной не совпадает с именем локальной переменной в области действия. В главе 16 мы обсудим применение *двухместной операции разрешения области действия* с классами.

На рис. 15.12 показана унарная операция разрешения области действия с локальной и глобальной переменными одного и того же имени. Чтобы подчеркнуть, что значения локальной и глобальной переменных **value** отличаются друг от друга, в программе одна из переменных объявляется с типом **float**, а другая с типом **int**.

```
// Использование унарной операции разрешения области действия
#include <iostream.h >

float value = 1.2345;

main ()
{
   int value = 7;

   cout << "Local value = " << value
        << "\nGlobal value = " << :: value << '\n' ;

   return 0 ;
}
```

```
Local value = 7
Global value = 1.2345
```

Рис. 15.12. Примечание унарной операции разрешения области действия

Распространенная ошибка программирования 15.18

Попытка обращения к не глобальной переменной во внешнем блоке с использованием унарной операции разрешения области действия.

Хороший стиль программирования 15.8

Избегайте объявления в программе переменных с одним и тем же именем для различных целей. Во многих случаях это является допустимым, но может создавать путаницу.

15.13. Перегрузка функций

В языке C объявление двух функций с одним и тем же именем в одной и той же программе является синтаксической ошибкой. C++ допускает определение нескольких функций с одним и тем же именем, пока эти функции различаются наборами параметров (по крайней мере их типами). Такая возможность называется *перегрузкой функций*. При вызове перегруженной функции компилятор C++ автоматически выбирает соответствующую функцию, исходя из анализа числа, типа и порядка параметров вызова. Перегрузка функций обычно используется для создания нескольких функций с одним и тем же именем, производящих схожие действия над различными типами данных.

Хороший стиль программирования 15.9

Перегруженные функции, выполняющие аналогичные задачи, делают программы более удобочитаемыми и понятными.

На рис. 15.13 перегруженная функция **square** используется для вычисления квадрата чисел типов **int** и **double**. В главе 17 мы обсудим, каким образом можно перегружать операции и специфицировать способ их воздействия на объекты определяемых пользователем типов. В разделе 15.15 вводятся шаблоны функций для выполнения идентичных задач над многими различными типами данных. В главе 20 подробно обсуждаются шаблоны функций и шаблоны классов.

```
// Использование перегруженных функций
#include <iostream.h >

int square(int x)   { return x * x; }

double square(double y)   { return y * y; }

main ()
{
   cout << "The square of integer 7 is "
        << square(7)
        << "\nThe square of double 7.5 is "
        << square(7.5) << '\n';

   return 0 ;
}
```

```
The square of integer 7 is 49
The square of double 7.5 is 56.25
```

Рис. 15.13. Использование перегруженных функций

Перегруженные функции различаются своей *сигнатурой* — комбинацией имени функции и типов ее параметров. Компилятор специальным образом кодирует идентификатор каждой функции (иногда это называют *декорированием имен*), используя количество и типы ее параметров, для обеспечения *безопасного по типу* редактирования связей (компоновки). Безопасное по типу редактирование связей гарантирует вызов соответствующей перегруженной

функции и должное согласование аргументов с параметрами. Компилятор выявляет ошибки компоновки и сообщает о них. Программа, представленная на рис. 15.14, транслировалась с помощью компилятора Borland C++. Закодированные имена функций на языке ассемблера, генерированные компилятором, показаны в окне вывода. Каждое декорированное имя начинается с символа @, за которым следует имя функции. Закодированный список параметров начинается с символов **$q**. В списке параметров функции **nothing2** **zc** представляет тип **char**, **i** представляет тип **int**, **pf** представляет **float *** и **pd** представляет **double ***. В списке параметров функции **nothing1** **i** представляет тип **int**, **f** представляет тип **float**, **zc** представляет тип **char** и **pi** представляет **int ***.

```
// Декорирование имен
int square(int x)  { return x * x; }

double square(double y)  { return y * y; }

void nothing1(int a, float b, char c, int *d)  {}

char *nothing2(char a, int b, float *c, double *d)  { return 0; }

main ()
{
   return 0;
}
```

```
public _main
public @nothing2$qzcipfpd
public @nothing1$qifzcpi
public @square$qd
public @square$qi
```

Рис. 15.14. Декорирование имен, обеспечивающее безопасное по типу редактирование связей

Две функции **square** различаются своими списками параметров; в одном **d** объявляется как **double**, а в другом **i** объявляется как **int**. Типы возвращаемых значений всех четырех функций в закодированных именах не специфицируются. Имейте в виду, что кодирование имен функций зависит от компилятора. Поэтому функция, компилированная в Borland C++, может иметь декорированное имя, отличное от того, которое она получила бы при использовании других компиляторов.

Перегруженные функции могут иметь разные типы возвращаемых значений, но при этом они все равно обязаны иметь различные списки параметров.

Распространенная ошибка программирования 15.19

Создание перегруженных функций с идентичными списками параметров и различными типами возвращаемых значений приводит к синтаксической ошибке.

Чтобы отличить функции с одним и тем же именем, компилятор использует только списки параметров. Перегруженные функции не обязаны иметь одно и то же количество параметров. Программисты должны проявлять осторожность при перегрузке функций с аргументами по умолчанию, поскольку это может приводить к неоднозначности.

Распространенная ошибка программирования 15.20

Функция с опущенными аргументами по умолчанию может вызываться точно так же, как другая перегруженная функция; это является синтаксической ошибкой.

Общее методическое замечание 15.2

Перегрузка функций позволяет устранить использование макросов **#define** языка С для выполнения идентичных операций над различными типами данных и обеспечивает строгую проверку типа аргументов, невозможную в макросах.

15.14. Спецификации внешней связи

Программы на C++ могут вызывать функции, написанные на С и компилировавшиеся с помощью компилятора С. Как говорилось в разделе 15.13, C++ специальным образом кодирует имена функций для безопасного по типу редактирования связей. Однако в языке С имена функций не кодируются. Таким образом, функция, компилированная в С, не будет распознана при попытке скомпоновать код на С с кодом на C++, поскольку код на C++ ожидает специально закодированного имени функции. C++ дает возможность программисту задать *спецификацию внешней связи*, которая информирует компилятор, что функция компилировалась с помощью компилятора С, и не допускают кодирования компилятором C++ имени функции. Спецификации внешней связи полезны в том случае, когда уже были разработаны большие библиотеки специализированных функций и пользователь или не имеет доступа к исходному тексту для перекомпиляции его на C++, или у него нет времени для преобразования библиотечных функций из С в C++.

Чтобы сообщить компилятору, что одна или несколько функций компилировались на С, запишите прототипы функций следующим образом:

```
extern "C" прототип функции // одна функция

extern "C" // несколько функций
{
   прототипы функций
}
```

Эти объявления сообщают компилятору, что указанные функции не компилировались на C++ и что поэтому для функций, перечисленных в спецификации внешней связи, кодирование имен выполняться не должно. В этом случае функции могут быть надлежащим образом скомпонованы с программой. Среда C++ обычно включает в себя стандартные библиотеки функций на С и не требует от программиста использования спецификаций внешней связи для этих функций.

15.15. Шаблоны функций

Перегруженные функции обычно используются для выполнения сходных операций над различными типами данных. Если операции для всех типов данных идентичны, то же самое можно сделать в более сжатой и удобной форме путем определения *шаблона функции*, средства, введенного в последних вер-

сиях C++. Программист пишет одно определение шаблона функции. Исходя из типа аргументов, задаваемых при вызовах этой функции, C++ автоматически генерирует объектный код для отдельных функций, обрабатывающих вызовы каждого типа. В языке C эту задачу можно выполнить с помощью макросов, создаваемых директивой **#define**. Однако макросы оставляют возможность для серьезных побочных эффектов и не позволяют компилятору осуществлять контроль соответствия типов. Шаблоны функций, подобно макросам, дают компактное решение проблемы, но при этом обеспечивают полную проверку типов.

Все определения шаблонов функций начинаются с ключевого слова **template**, за которым следует список формальных параметров шаблона функции, заключенный в угловые скобки (< и >). Каждому формальному параметру предшествует ключевое слово **class**. Эти формальные параметры используются, подобно встроенным типам или типам, определенным пользователем, для задания типов параметров функции, задания типа возвращаемого функцией значения и объявления переменных внутри функции. Далее следует определение шаблона, которое не отличается от определения любой другой функции.

Следующий шаблон функции используется в законченной программе на рис. 15.15.

```
template <class T>
void printArray(T* array, const int count)
{
   for (int i = 0; i < count; i++)
      cout << array[i] << " ";

   cout << '\n';
}
```

Этот шаблон функции объявляет единственный формальный параметр **T** в качестве типа массива, который выводится функцией **printArray**. Когда компилятор C++ находит в исходном коде программы вызов функции **printArray**, он подставляет тип первого параметра функции **printArray** вместо **T** по всему определению шаблона и создает законченный экземпляр функции для вывода массива данных с элементами указанного типа. Затем вновь созданная функция компилируется. На рис. 15.15 создаются экземпляры трех функций — одна из них принимает массив с элементами типа **int**, другая — массив с элементами типа **float** и третья — массив с элементами типа **char**. Реализация для типа **int** выглядит так:

```
void printArray(int *array, const int count)
{
   for (int i = 0; i < count; I++)
      cout << array[i] << " ";

   cout << '\n';
}
```

Каждый формальный параметр из заголовка шаблона должен появиться в списке параметров функции по крайней мере один раз. Имя параметра в списке формальных параметров шаблона может использоваться только один раз. Имена формальных параметров в различных шаблонах функций не обязаны быть уникальными.

Рис. 15.15 иллюстрирует использование шаблона функции **printArray**.

```
// Использование шаблонов функций
#include <iostream.h >

template <class T>
void printArray(T *array, const int count)
{
   for (int i = 0; i < count; i++)
      cout << array [i] << " ";

   cout << '\n';
}

main ()
{
   const int aCount = 5, bCount = 7, cCount = 6;
   int a [aCout] = {1, 2, 3, 4, 5};
   float b[bCount] = {1.1, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7};
   char c[cCount] = "HELLO";  // 6-я позиция для нуля

   cout << "Array a contains : \n";
   printArray(a, aCount);  // версия для целых чисел

   cout << "Array b contains: \n";
   printArray(b, bCount);  // версия для чисел с плавающей точкой

   cout << "Array c contains: \n";
   printArray(c, cCount);  // версия для символов

   return 0;
}
```

```
Array a contains:
1 2 3 4 5
Array b contains:
1.1 2.2 3.3 4.4 5.5 6.6 7.7
Array c contains:
H E L L O
```

Рис. 15.15. Использование шаблонов функций

Распространенная ошибка программирования 15.21

Отсутствие ключевого слова **class**, которое должно предшествовать каждому формальному параметру шаблона функции.

Распространенная ошибка программирования 15.22

Неиспользование какого-либо формального параметра шаблона в сигнатуре функции.

Резюме

- Язык C++ является надмножеством языка C, поэтому программисты могут использовать компилятор C++ для трансляции уже существующих программ на C и затем постепенно переводить эти программы на C++.

- Язык C требует, чтобы комментарий был заключен между символами /* и */. C++ допускает комментарии, начинающиеся с // и завершаемые концом строки.
- Язык C++ допускает размещение объявлений всюду, где может находиться исполняемый оператор. Объявления переменных могут также включаться в раздел инициализации структуры **for**.
- Язык C++ предусматривает альтернативную обращениям к функциям **printf** и **scanf** возможность для обработки ввода/вывода стандартных типов данных и строк.
- Выходной поток **cout** и операция << (операция передачи в поток, произносится «послать в») применяются для вывода данных. Входной поток **cin** и операция >> (операция извлечения из потока, произносится «взять из») обеспечивают ввод данных.
- Язык C++ предоставляет программисту возможность создавать определяемые пользователем типы с помощью ключевых слов **enum**, **struct**, **union** и **class**. Имена-этикетки перечислений, структур, объединений или классов могут впоследствии использоваться для определения переменных нового типа.
- Программы не компилируются, если для каждой функции не задан ее прототип или функция не определена до обращения к ней (в этом случае отдельного прототипа функции не требуется).
- Функция, которая не возвращает значения, объявляется с типом возвращаемого значения **void**. При попытке возвратить значение из этой функции или использовать результат ее вызова в вызывающей функции компилятор генерирует ошибку.
- В языке C++ пустой список параметров специфицируется либо пустыми круглыми скобками, либо записью в них слова **void**. В C в случае пустого списка параметров отключается проверка аргументов функции.
- Встроенные функции исключают накладные расходы, связанные с вызовом функций. Программист, используя ключевое слово **inline**, советует компилятору генерировать код функции на месте ее вызова (если это возможно), чтобы минимизировать обращения к функциям. Компилятор может игнорировать спецификатор **inline**.
- Язык C++ предусматривает явную форму передачи параметров по ссылке — с использованием параметров-ссылок. Чтобы указать, что параметр функции передается по ссылке, поместите после типа этого параметра в прототипе функции символ амперсанда &. При вызове функции укажите просто имя переменной, и она будет передана по ссылке. В вызываемой функции ссылка на локальное имя этой переменной на самом деле относится к исходной переменной в вызывающей функции. Таким образом, исходная переменная может непосредственно модифицироваться вызываемой функцией.
- Переменные-ссылки могут также создаваться для локального использования внутри некоторой функции в качестве псевдонимов других переменных. Переменные-ссылки должны инициализироваться при их объявлении и не могут быть переназначены в качестве псевдонимов других переменных. Как только ссылка объявлена в качестве псевдонима пере-

менной, все операции, выполняемые якобы над псевдонимом, на самом деле выполняются над этой переменной.

- При помощи модификатора **const** можно создавать «константные переменные». Константная переменная должна быть инициализирована константным выражением при ее объявлении и не может быть изменена в дальнейшем. Константные переменные часто называют именованными константами или переменными только для чтения. Константные переменные могут использоваться всюду, где ожидается константное выражение. Другими распространенными случаями применения модификатора **const** являются константные указатели, указатели на константы и константные ссылки.
- Операции **new** и **delete** языка C++ предоставляют программисту более изящный способ управления динамическим распределением памяти, чем функции **malloc** и **free** языка С. Операция **new** принимает тип в качестве операнда, автоматически создает объект соответствующего размера и возвращает указатель правильного типа. Операция **delete** принимает в качестве операнда указатель на объект, созданный операцией **new**, и освобождает память.
- Язык C++ позволяет программисту специфицировать аргументы по умолчанию и значения по умолчанию для этих аргументов. Если аргумент по умолчанию при вызове функции опускается, то используется значение по умолчанию. Аргументы по умолчанию должны быть самыми правыми (последними) параметрами в списке параметров функции. Аргументы по умолчанию должны объявляться при первом появлении имени функции.
- Унарная операция разрешения области действия (::) дает возможность программе обращаться к глобальной переменной, когда в области действия находится локальная переменная с тем же именем.
- Возможно определение нескольких функций с одним и тем же именем, но с различными наборами параметров. Это называется перегрузкой функций. При вызове перегруженной функции компилятор автоматически выбирает соответствующую функцию путем анализа числа и типа аргументов вызова.
- Перегруженные функции могут иметь разные типы возвращаемых значений, но они обязаны иметь различные списки параметров. Объявление двух функций, различающихся только типами возвращаемых значений, вызовет ошибку компиляции.
- В некоторых ситуациях необходимо вызывать функции, написанные на С и транслированные с помощью компилятора С (например, функции специализированной библиотеки компании, которые не были преобразованы в C++ и перекомпилированы). В C++ имена функций обрабатываются иначе, чем в С. Таким образом, функция, компилированная в С, не будет распознана при попытке скомпоновать код на С с кодом на C++. Язык C++ предусматривает спецификацию внешней связи, информирующую компилятор, что функция транслировалась с помощью компилятора С, и ее имя не должно кодироваться подобно имени функции C++.

- Шаблоны функций дают возможность порождать функции, которые выполняют одинаковые операции над различными типами данных, однако сам шаблон функции при этом определяется только один раз.

Терминология

<iostream.h>
asm
catch
cin (входной поток)
class
const
cout (выходной поток)
friend
inline — (встроенная) функция
template
this
throw
try
virtual
аргументы функции по умолчанию
безопасная по типу компоновка
библиотека потоков
динамические объекты
именованная константа
инициализатор
класс
ключевое слово **operator**
область свободной памяти
однострочный комментарий (//)
операция «взять из» (>)
операция «послать в» (<)
операция **delete**
операция **new**
операция извлечения из потока (>)
операция передачи в поток (<)
параметр-ссылка
перегрузка
перегрузка функций
переменная «только для чтения»
переменная-константа
сигнатура
спецификация внешней связи
суффикс-амперсанд (&)
типы ссылки
унарная операция разрешения области действия (::)
шаблон
шаблон функции

Распространенные ошибки программирования

15.1. Забывают закрыть комментарий в стиле C символами */.

15.2. Объявление переменной после ссылки на нее в каком-либо операторе.

15.3. Попытка возвратить значение из функции типа **void** или использовать результат вызова функции **void**.

15.4. Программы на C++ не компилируются, если для каждой функции не объявлен ее прототип или она не определена до обращения к ней.

15.5. C++ является развивающимся языком, и некоторые его особенности могут не поддерживаться вашим компилятором. Использование нереализованных свойств языка приводит к синтаксическим ошибкам.

15.6. Поскольку доступ к параметрам-ссылкам в теле вызываемой функции производится просто по имени, программист может по неосторожности обращаться с параметрами-ссылками как с параметрами, передаваемыми по значению. Это может привести к неожиданным побочным эффектам, если исходные копии переменных будут изменены вызывающей функцией.

15.7. Отсутствие инициализации переменной-ссылки при ее объявлении.

15.8. Попытка переназначить ранее объявленную ссылку в качестве псевдонима другой переменной.

15.9. Попытка разыменования переменной-ссылки с помощью операции косвенной адресации (не забывайте, что ссылка является псевдонимом для переменной, а не указателем на нее).

15.10. Возвращение указателя или ссылки на автоматическую переменную, определенную в вызываемой функции.

15.11. Использование переменных с модификатором **const** в объявлениях массивов и помещение переменных с модификатором **const** в заголовочные файлы (включаемые в различные исходные файлы одной и той же программы) запрещено в С, но разрешено в С++.

15.12. Инициализация переменной **const** неконстантным выражением, например, переменной, не объявленной с модификатором **const**.

15.13. Попытка изменить константную переменную.

15.14. Попытка изменить константный указатель.

15.15. Освобождение с помощью **delete** памяти, которая не была выделена операцией **new**.

15.16. Применение операции **delete** без скобок ([и]) при удалении динамически созданных массивов (это существенно только для массивов, содержащих объекты пользовательских типов).

15.17. Определение и попытка использовать аргумент по умолчанию, который не является самым правым (последним) аргументом (если одновременно не являются аргументами по умолчанию все аргументы справа от него).

15.18. Попытка обращения к не глобальной переменной во внешнем блоке с использованием унарной операции разрешения области действия.

15.19. Создание перегруженных функций с идентичными списками параметров и различными типами возвращаемых значений приводит к синтаксической ошибке.

15.20. Функция с опущенными аргументами по умолчанию может вызываться точно так же, как другая перегруженная функция; это является синтаксической ошибкой.

15.21. Отсутствие ключевого слова **class**, которое должно предшествовать каждому формальному параметру шаблона функции.

15.22. Неиспользование какого-либо формального параметра шаблона в сигнатуре функции.

Хороший стиль программирования

15.1. Использование комментариев в стиле языка С++ с // помогает избегать ошибок незавершенного комментария, которые возникают при пропуске символов */, закрывающих комментарии С.

15.2. Применение ориентированного на потоки ввода/вывода в стиле языка С++ делает программы более удобочитаемыми (и менее уяз-

вимыми для ошибок), чем их эквиваленты на C с вызовами функций **printf** и **scanf**.

15.3. Размещение объявлений переменных поблизости от их первого использования может сделать программу более ясной.

15.4. Модификатор **inline** следует использовать только с небольшими, часто используемыми функциями.

15.5. Используйте указатели для передачи параметров, которые могут быть изменены вызываемой функцией, и используйте ссылки на константы для передачи параметров большого размера, не подлежащих изменению.

15.6. Преимуществом использования переменных с модификатором **const** вместо символических констант является то, что переменные с модификатором **const** видимы для символического отладчика; константы, определяемые с помощью **#define**, для него невидимы.

15.7. Задание аргументов по умолчанию может упростить отдельные вызовы функций. Однако некоторые программисты считают, что явное указание всех аргументов вызова делает программу более понятной.

15.8. Избегайте объявления в программе переменных с одним и тем же именем для различных целей. Во многих случаях это является допустимым, но может создавать путаницу.

15.9. Перегруженные функции, выполняющие аналогичные задачи, делают программы более удобочитаемыми и понятными.

Советы по повышению эффективности

15.1. Использование встроенных функций может привести к уменьшению времени выполнения программы, но при этом увеличить ее размер.

15.2. Для больших объектов используйте параметр-ссылку с **const**, чтобы имитировать внешний вид и надежность передачи параметра по значению, но избежать накладных расходов, связанных с передачей копии большого объекта.

Советы по переносимости программ

15.1. Смысл пустого списка параметров функции в языке C++ разительно отличается от его смысла в C. В C это означает, что блокируется любая проверка параметров. В C++ это означает, что функция не принимает параметров. Поэтому программы на C, использующие такие объявления, могут выполняться иначе при компиляции в C++.

Общие методические замечания

15.1. Любое изменение встроенной функции требует перекомпиляции всех функций, являющихся ее клиентами. Это может быть существенным в определенных ситуациях, связанных с разработкой и сопровождением программ.

15.2. Перегрузка функций позволяет устранить использование макросов **#define** языка С для выполнения идентичных операций над различными типами данных и обеспечивает строгую проверку типа аргументов, невозможную в макросах.

Упражнения для самоконтроля

15.1. Заполните пробелы в каждом из следующих предложений:

a) В языке С ++ можно объявить несколько функций с одним и тем же именем, но различными типами и/или числом аргументов. Это называется _____ функций.

b) _____ дает возможность обращаться к глобальной переменной с тем же именем, что и переменная в текущей области действия.

c) Операция _____ динамически выделяет память для нового объекта.

d) Предположим, что в С **a** и **b** являются целочисленными переменными и мы образуем сумму **a + b**. Теперь предположим, что **c** и **d** являются переменными с плавающей точкой и мы образуем сумму **c + d**. Очевидно, что две операции + здесь используются для различных целей. Это — пример свойства, имеющегося в языке С ++, которое также имеется в С и называется _____ операций.

e) Двумя предопределенными в С++ потоковыми объектами, которые мы обсуждали, являются _____ и _____.

f) Ключевые слова _____, _____, _____ и _____ используются для создания новых типов данных в языке С++.

g) Модификатор _____ используется для объявления переменных только для чтения.

h) В С++ возможна _____, чтобы функции, компилированные с помощью компилятора С, могли надлежащим образом компоноваться с программой на С++.

i) _____ функций дают возможность определить одну функцию для выполнения некоторых действий над различными типами данных.

15.2. (Верно/неверно) При написании комментария в виде блока, для которого требуется много строк текста, более экономным является использование разделителя комментария С++ //, чем обычных разделителей С /* и */.

15.3. Почему в языке С++ прототип функции может содержать объявление типа параметра в виде **float&**?

15.4. (Верно/неверно) Все вызовы в С++ выполняются с передачей параметров по значению.

15.5. Объясните, почему операции << и >> в языке С++ являются перегруженными.

15.6. Напишите фрагменты программ на C++ для выполнения каждой из следующих задач:

a) Запишите строку **«Welcome to C!»** в стандартный выходной поток **cout**.

b) Прочитайте значение переменной **age** из стандартного входного потока **cin**.

15.7. Напишите законченную программу на C++, которая, используя потоковый ввод/вывод и встроенную функцию **sphereVolume**, запрашивает у пользователя значение радиуса шара, а затем вычисляет (и выводит) объем этого шара по формуле **volume = (4/3) PI * pow(radius, 3)**.

15.8. Что бы произошло, если бы в языке C вы дважды объявили одну и ту же функцию с различными типами параметров? Что бы произошло в языке C++?

Ответы к упражнениям для самоконтроля

15.1. a) перегрузкой. b) Унарная операция разрешения области действия (::). c) **new**. d) перегрузкой. e) **cin**, **cout**. f) **enum**, **struct**, **union**, **class**. g) **const**. h) спецификация внешней связи. i) Шаблоны.

15.2. Неверно. Каждый из обычных разделителей в стиле языка C необходимо написать один раз, в то время как разделитель // языка C++ должен появляться в начале каждой новой строки.

15.3. Потому что программист объявляет параметр-ссылку типа «ссылка на» **float**, чтобы посредством передачи параметра по ссылке получить доступ к исходной переменной — аргументу.

15.4. Неверно. C++ допускает настоящую передачу по ссылке посредством параметров-ссылок в дополнение к использованию для этой цели указателей.

15.5. Операция >> является в зависимости от контекста либо операцией сдвига вправо, либо операцией извлечения из потока. Операция << является в зависимости от контекста либо операцией сдвига влево, либо операцией передачи в поток.

15.6. a)
```
cout << "Welcome to C!";
```

b)
```
cin >> age;
```

15.7.
```
// Встроенная функция, вычисляющая объем шара
#include <iostream.h>

const float PI = 3.14159;

inline float sphereVolume(const float r) {return
                              4:0 / 3.0 * PI * r * r * r;}
main()
{
   float radius;
   cout << "Enter the length of the radius
                              of your sphere:";
   cin >> radius;
```

```
   cout << "Volume of sphere with radius " << radius <<
           "is " << sphereVolume(radius) << '\n';
   return 0;
}
```

15.8. В языке C произошла бы ошибка компиляции, означающая, что двум функциям было присвоено одно и то же имя. В языке C++ это является примером перегрузки функций — что допустимо, — и никакой ошибки не возникло бы.

Упражнения

15.9. Предположим, что некая организация в настоящее время делает в программировании основной упор на языке C и хотела бы перейти к программированию в объектно-ориентированной среде C++. Какая стратегия подходит для перехода от среды программирования C к среде C++?

15.10. Напишите несколько поточно-ориентированных операторов в стиле C++ для выполнения каждой из следующих задач:

a) Отобразите на экране **«HELLO»**.

b) Введите с клавиатуры значение для переменной **temperature** типа **float**.

15.11. Сравните ввод/вывод в стиле **printf/scanf** с потоковым вводом/выводом в стиле языка C++.

15.12. В этой главе мы упоминали, что существует много случаев, когда C++ автоматически «знает», какие типы необходимо использовать. Перечислите как можно больше таких ситуаций.

15.13. Напишите программу на C++, которая использует потоковый ввод/вывод для запроса у пользователя семи целых чисел, а также определяет и выводит их максимум.

15.14. Напишите законченную программу на C++, которая использует потоковый ввод/вывод и встроенную функцию **circleArea** для запроса у пользователя радиуса круга, а также вычисления и вывода площади этого круга.

15.15. Напишите законченную программу на C++ с тремя альтернативными функциями, описанными ниже, каждая из которых просто добавляет 1 к переменной **count**, определенной в функции **main**. Затем сопоставьте три альтернативных подхода. Эти три функции таковы

a) Функция **add1CallByValue**, которая получает копию переменной **count** по значению, прибавляет 1 к этой копии и возвращает новое значение.

b) Функция **add1ByPointer**, в которой доступ к переменной **count** передается посредством имитации ссылки с использованием указателя и применяется операция разыменования * для прибавления 1 к исходной копии переменной **count** в **main**.

c) Функция **add1ByReference**, в который происходит подлинная передача по ссылке переменной **count** посредством параметра-ссылки

и 1 прибавляется к исходной копии переменной **count** с использованием ее псевдонима (т.е. параметра-ссылки).

15.16. Для чего используется унарная операция разрешения области действия?

15.17. Сравните динамическое распределение памяти с помощью операций C++ **new** и **delete** с динамическим распределением памяти посредством функций **malloc** и **free** стандартной библиотеки языка C.

15.18. Перечислите различные свойства языка C++ из представленных нами в этой главе, которые представляют собой усовершенствования языка C и не связаны с объектно-ориентированным программированием.

15.19. Напишите программу, которая использует шаблон функции с именем **min** для определения наименьшего из двух аргументов. Протестируйте программу, используя пары чисел типов **int**, **char** и **float**.

15.20. Напишите программу, которая использует шаблон функции с именем **max** для определения наибольшего из трех аргументов. Протестируйте программу, используя тройки чисел типов **int**, **char** и **float**.

15.21. Определите, имеются ли ошибки в следующих фрагментах программы. Для каждой ошибки покажите, как ее исправить. Замечание: возможно, что в том или ином конкретном фрагменте программы ошибок нет.

a)
```
main ()
{
   cout << x;
   int x = 7;
   return 0;
}
```

b)
```
template <class A>
int sum(int num1, int num2, int num3)
{
   return num1 + num2 + num3;
}
```

c)
```
/* Это комментарий
```

d)
```
void printResults(int x, int y)
{
   cout << "The sum is " << x + y << '\n';
   return x + y;
}
```

e)
```
char *s = "character string";
delete s;
```

f)
```
float &ref;
*ref = 7;
```

g)
```
int x;
const int y = x;
```

h)
```
template <A>
A product(A num1, A num2, A num3)
{
   return num1 * num2 * num3;
}
```

i)
```
const double pi = 3.14159;
pi = 3;
```

j)
```
// Это комментарий
```

k)
```
char *s = new char[10];
delete s;
```

l)
```
double cube(int);
int cube(int);
```

глава

16

Классы и абстракция данных



Цели

- Понять принципы инкапсуляции и сокрытия данных при конструировании программного обеспечения.
- Усвоить понятия абстракции данных и абстрактных типов (ADT).
- Научиться создавать абстрактные типы данных C++, а именно классы.
- Изучить создание, использование и уничтожение объектов класса.
- Изучить управление доступом к элементам данных и функциям объектов.
- Получить представление о преимуществах ориентации на объекты.

Содержание

16.1. Введение
16.2. Определение структур
16.3. Доступ к элементам структур
16.4. Реализация пользовательского типа Time с помощью структуры
16.5. Реализация абстрактного типа данных Time с помощью класса
16.6. Область действия класса и доступ к элементам класса
16.7. Отделение интерфейса от реализации
16.8. Управление доступом к элементам класса
16.9. Функции доступа и сервисные функции
16.10. Инициализация объектов класса: конструкторы
16.11. Использование с конструкторами параметров по умолчанию
16.12. Деструкторы
16.13. Когда вызываются конструкторы и деструкторы
16.14. Использование элементов данных и элементов-функций
16.15. Скрытая ловушка: возвращение ссылки на закрытый элемент данных
16.16. Присваивание по умолчанию путем поэлементного копирования
16.17. Повторное использование программного обеспечения

Резюме • Распространенные ошибки программирования • Хороший стиль программирования • Советы по повышению эффективности • Общие методические замечания • Упражнения для самоконтроля • Ответы к упражнениям для самоконтроля • Упражнения

16.1. Введение

Теперь мы вплотную подходим к концепции ориентации на объекты. Мы увидим, что объектная ориентация является естественным способом восприятия мира и написания компьютерных программ. Почему же в таком случае мы не начали с этого прямо на первой странице этой книги? Почему мы отложили объектно-ориентированное программирование на C++ до главы 16? Ответ состоит в том, что объекты, которые мы будем строить, частично будут состоять из фрагментов структурных программ, поэтому сначала нам нужно было заложить фундамент в виде структурного программирования.

В первых главах мы сосредоточились на «традиционных» методах структурного программирования. В этой главе мы представляем базовые понятия (т.е. «мышление объектами») и терминологию (т. е. «разговор на языке объектов») объектно-ориентированного программирования. В последующих главах мы рассмотрим более существенные вопросы и подойдем к решению сложных задач при помощи методов *объектно-ориентированного проектирования (OOD)*: мы проанализируем типичные постановки задач, которых требует построение программных систем, определим, какие объекты необходимы для реализации этих систем, какие атрибуты потребуются для этих объектов, каким должно быть поведение этих объектов, и зададим способы взаимодействия объектов друг с другом для достижения общесистемных целей.

Мы начнем с представления некоторых ключевых терминов объектной ориентации. Посмотрите на реальный мир вокруг вас. Куда бы вы ни посмотрели, всюду вы увидите их — *объекты*! Люди, животные, растения, автомобили, самолеты, здания, газонокосилки, компьютеры и тому подобное. Люди мыслят представлениями объектов. Мы обладаем замечательной способностью *абстракции*, которая позволяет нам рассматривать изображение на экране в виде объектов, как, например, люди, растения, деревья и горы, а не отдельных цветных точек. Мы можем, если захотим, мыслить представлениями пляжей, а не песчинок, леса, а не деревьев, и домов, а не кирпичей.

Может быть, нам покажется разумным разделить объекты на две категории — живые и неживые объекты. Живые объекты в определенном смысле «действуют». Они движутся и что-то делают. Неживые объекты, вроде полотенец, кажется, мало что делают. Они только в некотором роде «создают фон». Однако все эти объекты имеют немало общего. Все они обладают *атрибутами*, вроде размера, формы, цвета, веса и т.п. И все они демонстрируют разнообразное *поведение*, например, мяч катится, подпрыгивает, его надувают и спускают; ребенок кричит, спит, ползает, ходит и моргает; автомобиль разгоняется, тормозит, поворачивает и т.д.

Люди познают объекты, изучая их свойства и наблюдая за их поведением. Различные объекты могут иметь много одинаковых свойств и демонстрировать сходное поведение. Можно, например, сравнивать детей и взрослых, людей и шимпанзе. Легковые автомобили, грузовики, красные фургончики и роликовые коньки имеют много общего.

Объектно-ориентированное программирование (OOP) моделирует объекты реального мира при помощи их программных эквивалентов. Оно использует понятие *класса*, когда объекты некоторого класса — например, класса средств передвижения — имеют одни и те же характеристики. Оно использует отношения *наследования* и даже *сложного наследования*, когда вновь создаваемые классы объектов получаются путем наследования характеристик существующих классов и при этом содержат, однако, свои собственные уникальные атрибуты. Дети наследуют многие особенности своих родителей, однако иногда у низкорослых родителей бывают высокие дети.

Объектно-ориентированное программирование дает нам возможность подходить к процессу создания программ более естественно и интуитивно, а именно путем *моделирования* объектов реального мира, их свойств и их поведения. OOP также моделирует коммуникацию между объектами. Точно так же, как люди посылают друг другу *сообщения* (например, сержант, дающий команду новобранцам стать по стойке смирно), так и объекты взаимодействуют посредством сообщений.

ООР *инкапсулирует* данные (атрибуты) и функции (поведение) в пакетах, называемых *объектами*; данные и функции объекта тесно связаны друг с другом. Объекты обладают свойством *сокрытия информации*. Это означает, что хотя объекты могут знать о способах взаимодействия друг с другом посредством точно определенного интерфейса, обычно они не располагают информацией о способах реализации других объектов — подробности реализации сокрыты внутри самих объектов. Не вызывает сомнения, что можно эффективно управлять автомобилем и не зная подробностей внутренней работы двигателя, коробки передач и выхлопной системы. Мы увидим, почему сокрытие информации является столь критичным для систематической разработки программного обеспечения.

В языке С и других *процедурных языках программирования* существует тенденция к *ориентации на действие*, в то время как программирование на С++ имеет тенденцию к *объектной ориентации*. В С программной единицей является *функция*. В С++ программной единицей является *класс*, на основе которого в конечном итоге создаются *представители* класса, т.е. *экземпляры* объектов.

Программирующие на С концентрируют внимание на написании функций. Совокупность действий, выполняющих некоторую общую задачу, оформляется в виде функции, а совокупность функций образует программу. Данные, безусловно, важны в языке С, однако имеет место точка зрения, что данные существуют прежде всего для поддержки выполняемых функциями действий. *Глаголы* в спецификации системы помогают программисту определить набор функций, совместная работа которых обеспечит ее реализацию.

Программирующие на С++ концентрируются на определении своих собственных *пользовательских типов*, называемых классами. Каждый класс содержит как данные, так и набор функций, манипулирующих этими данными. Данные класса, входящие в его состав, называются *элементами данных* класса. Функциональные компоненты класса называются его *элементами-функциями*. Если представитель встроенного типа, например, типа **int**, называется *переменной*, то представитель пользовательского типа (т.е. класса) называется *объектом*. Внимание в языке С++ фокусируется на объектах, а не на функциях. *Существительные* в спецификации системы помогают программисту определить набор классов С++, на основе которых будут созданы объекты, совместная работа которых обеспечит реализацию системы.

Классы в С++ являются естественным развитием понятия **struct** языка С. Прежде чем перейти к специальным вопросам разработки классов в С++, мы еще раз рассмотрим структуры и создадим пользовательский тип, основанный на структуре. Слабости, которые мы выявим при этом подходе, помогут нам обосновать понятие класса.

16.2. Определение структур

Рассмотрим следующее определение структуры:

```
struct Time {
   int hour;    // 0-23
   int minute;  // 0-59
   int second;  // 0-59
};
```

Давайте повторим терминологию, связанную со структурами (см. главы 10 и 12). Ключевое слово **struct** начинает определение структуры. Идентификатор **Time** называется *этикеткой структуры*. Этикетка структуры именует ее определение и используется для объявления переменных *структурного типа*. В этом примере именем структурного типа является **Time** (в противоположность более длинному имени **struct Time**, обязательному в C). Имена, объявленные в фигурных скобках в области определения структуры, называются *элементами* структуры. Элементы одной и той же структуры должны иметь уникальные имена, однако две различные структуры могут содержать элементы с одним и тем же именем без возникновения конфликта. Каждое определение структуры должно заканчиваться точкой с запятой. Как мы скоро увидим, все вышесказанное применимо и для классов.

Определение структуры **Time** содержит три элемента типа **int** — **hour**, **minute** и **second**. Элементы структуры могут иметь любой тип. Структура не может, однако, содержать экземпляр себя самой. Например, элемент типа **Time** не может быть объявлен в определении структуры **Time**. Однако в нее может быть включен указатель на структуру **Time**. Структура, содержащая элемент, являющийся указателем на тот же самый структурный тип, называется *ссылающейся на себя* структурой. Такие структуры полезны при построении связанных структур данных (см. главу 12).

Предыдущее определение структуры не резервирует места в памяти; это определение создает новый тип данных, который используется для объявления переменных. Структурные переменные объявляются подобно переменным других типов. В объявлении

```
Time timeObject, timeArray[10], *timePtr;
```

timeObject объявляется переменной типа **Time**, **timeArray** массивом из 10 элементов типа **Time** и **timePtr** указателем на объект **Time**.

16.3. Доступ к элементам структур

Для доступа к элементам структуры (или класса) используются *операции доступа к элементам* — *операция-точка* (.) и *операция-стрелка* (->). Операция-точка обращается к элементу структуры (или класса) через имя переменной объекта или ссылку на объект. Например, чтобы вывести элемент hour структуры **timeObject**, используйте оператор

```
cout << timeObject.hour;
```

Операция-стрелка, состоящая из знака минус (-) и знака больше (>) без пробелов между ними, обращается к элементу структуры (или класса) через указатель на объект. Предположим, что для обращения к объектам **Time** был объявлен указатель **timePtr** и что ему был присвоен адрес структуры **timeObject**. Чтобы вывести элемент **hour** структуры **timeObject** при помощи указателя **timePtr**, используйте оператор

```
cout << timePtr->hour;
```

Выражение **timePtr->hour** эквивалентно выражению **(*timePtr).hour**, в котором происходит разыменование указателя и обращение к элементу **hour** при помощи операции-точки. Круглые скобки здесь необходимы, поскольку операция-точка (.) имеет более высокий приоритет, чем операция разыменования указателя (*). Операция-стрелка и операция элемента структуры, наряду

с круглыми и квадратными скобками ([]), имеют самый высокий приоритет (из рассмотренных до сих пор операций) и ассоциируются слева направо.

16.4. Реализация пользовательского типа Time с помощью структуры

На рис. 16.1 создается пользовательский структурный тип **Time** с тремя целочисленными элементами: **hour**, **minute** и **second**. В программе определяется одна структура **Time** с именем **dinnerTime** и используется операция-точка для инициализации элемента структура **hour** значением **18**, **minute** значением **30** и **second** значением **0**. Затем программа выводит время в военном и стандартном форматах. Обратите внимание, что функции **print...** получают ссылки на константные структуры **Time**. Это приводит к передаче структур **Time** в функции **print...** по ссылке — исключая, таким образом, непроизводительные затраты на копирование, связанные с передачей структур функциям по значению, а также предотвращает изменение структуры **Time** функциями **print...**. В главе 17 мы обсудим объекты и элементы-функции с модификатором **const**.

```
// FIG16_1. CPP
// Создает структуру, устанавливает и выводит ее элементы.
#include <iostream.h>

struct Time {    // определение структуры
   int hour;     // 0-23
   int minute;   // 0-59
   int second;   // 0-59
};

void printMilitary(const Time &);  // прототип
void printStandard(const Time &);  // прототип

main ()
{
   Time dinnerTime;     // переменная нового типа Time

   // присваивает элементам допустимые значения
   dinnerTime.hour = 18;
   dinnerTime.minute = 30;
   dinnerTime.second = 0;

   cout << "Dinner will be held at ";
   printMilitary(dinnerTime);
   cout << " military time,\nwhich is " ;
   printStandard(dinnerTime);
   cout << " standart time." << endl;

   // присваивает элементам недопустимые значения
   dinnerTime.hour = 29;
   dinnerTime.minute = 73;
   dinnerTime.second = 103;
```

Рис. 16.1. Создание структуры, инициализация и вывод ее элементов (часть 1 из 2)

```
   cout << "\nTime with invalid values: ";
   printMilitary(dinnerTime);
   cout << endl;
   return 0;
}

// Выводит время в военном формате
void printMilitary(const Time &t)
{
   cout << (t.hour < 10 ? "0" : "" ) << t.hour << ":"
        << (t.minute < 10 ? "0" : "" ) << t.minute << ":"
        << (t.second < 10 ? "0" : "" ) << t.second;
}

// Выводит время в стандартном формате
void printStandard(const Time &t)
{
   cout << ((t.hour == 0 || t.hour == 12) ? 12 : t.hour % 12)
        << ":" << (t.minute < 10 ? "0" : "") << t.minute
        << ":" << (t.second < 10 ? "0" : "") << t.second
        << (t.hour < 12 ? " AM" : " PM");
}
```

```
Dinner will be held at 18:30:00 military time,
which is 6:30:00 PM standard time.

Time with invalid values:  29:73:103
```

Рис. 16.1. Создание структуры, инициализация и вывод ее элементов (часть 2 из 2)

Совет по повышению эффективности 16.1

Обычно структуры передаются по значению. Чтобы избежать накладных расходов на их копирование, передавайте структуры по ссылке.

Совет по повышению эффективности 16.2

Чтобы избежать накладных расходов при передаче параметров по значению и вместе с тем извлечь выгоду из защиты от изменения исходных данных, передавайте параметры большого размера в виде константных ссылок.

Существуют и отрицательные стороны создания новых типов данных с помощью структур. Поскольку инициализация специально не оговаривается, то данные могут оказаться неинициализированными и вследствие этого могут возникнуть проблемы. Даже если данные инициализированы, может оказаться, что они инициализированы некорректно. Элементам структуры могут быть присвоены недопустимые значения (как мы это сделали на рис. 16.1), поскольку программа имеет прямой доступ к данным. Программа присвоила некорректные значения всем трем элементам объекта **dinnerTime** типа **Time**. При изменении реализации **struct** все программы, использующие эту **struct**, должны быть изменены. Это происходит потому, что программист непосредственно манипулирует типом данных. Не существует никакого «интерфейса», позволяющего гарантировать, что программист корректно использует тип данных и что данные остаются действительными.

Общее методическое замечание 16.1

Важно писать понятные и простые для сопровождения программы. Изменения в программах являются скорее правилом, чем исключением. Программисты должны предвидеть заранее, что их код будет модифицироваться. Как мы увидим, классы способствуют модифицируемости программ.

Существуют и другие проблемы, связанные со структурами в стиле языка C. Эти структуры нельзя вывести как единое целое, напротив, их элементы должны выводиться и форматироваться по одному. Для вывода элементов структуры в подходящем формате может быть написана специальная функция. В главе 18, «Перегрузка операций», показано, как можно перегрузить операцию <<, чтобы реализовать вывод объектов типа структуры или класса в удобной форме. Структуры нельзя сравнивать как единое целое; они должны сравниваться поэлементно. В главе 18 также показано, как можно перегрузить операции равенства и отношений для сравнения объектов типа структуры и класса.

В следующем разделе мы повторно реализуем нашу структуру **Time** в виде класса и продемонстрируем некоторые преимущества создания абстрактных типов данных с помощью классов. Мы увидим, что в C++ классы и структуры могут использоваться практически идентично. Различие между ними состоит в доступе по умолчанию к их элементам. К этому мы вскоре вернемся.

16.5. Реализация абстрактного типа данных Time с помощью класса

Классы дают возможность программисту моделировать объекты, которые обладают *атрибутами* (представленными в виде элементов данных) и которым присуще определенное *поведение* или *действие* (представленное в виде элементов-функций). При определении типов, содержащих элементы данных и элементы-функции, в C++ обычно используется ключевое слово **class**.

В других объектно-ориентированных языках программирования элементы-функции иногда называют *методами* и они активируются в ответ на посылаемые объекту *сообщения*. Сообщение соответствует вызову элемента-функции.

Сразу после определения класса его имя может быть использовано для объявления объектов этого класса. На рис. 16.2 показано простое определение класса **Time**.

```
class Time {
public:
   Time();
   void setTime(int, int, int);
   void printMilitary();
   void printStandard();
private:
   int hour;      // 0 - 23
   int minute;    // 0 - 59
   int second;    // 0 - 59
};
```

Рис. 16.2. Простое определение класса **Time**

Определение нашего класса **Time** начинается с ключевого слова **class**. *Тело* определения класса ограничено левой и правой фигурными скобками ({ и }). Определение класса завершается точкой с запятой. В определении нашего класса **Time**, как и в определении структуры **Time**, содержится три целочисленных элемента **hour**, **minute** и **second**.

Распространенная ошибка программирования 16.1

Забывают о точке с запятой в конце определения класса.

Остальные компоненты определения класса являются новыми. Метки **public:** и **private:** называются *спецификаторами доступа к элементам*. Все элементы данных и элементы-функции, объявленные после спецификатора **public:** (и до следующего спецификатора доступа) доступны всюду, где программа имеет доступ к какому-либо объекту класса **Time**. Все элементы данных и элементы-функции, объявленные после спецификатора **private:** (и до следующего спецификатора доступа) доступны только для функций-элементов класса. Спецификаторы доступа к элементам класса всегда заканчиваются двоеточием (:) и могут многократно появляться в определении класса.

Хороший стиль программирования 16.1

Используйте в определении класса каждый спецификатор доступа только один раз из соображений ясности и удобочитаемости программы. Помещайте открытые элементы в начале определения, чтобы их было проще найти.

Определение класса после спецификатора **public:** содержит прототипы следующих четырех элементов-функций: **Time**, **setTime**, **printMilitary** и **printStandard**. Эти функции называют *открытыми*, или *публичными* элементами-функциями, а также *интерфейсом* класса. Эти функции будут использоваться *клиентами* (пользователями) класса для манипулирования его данными.

Элемент-функция с тем же именем, что и сам класс, называется *конструктором* этого класса. Конструктор — это специальная функция-элемент класса, которая инициализирует элементы данных объекта класса. Конструктор класса вызывается автоматически при создании объекта.

Три целочисленных элемента появляются после спецификатора **private:**. Это означает, что эти элементы данных доступны только для функций-элементов класса — и, как мы увидим в следующей главе, для друзей класса. Таким образом, к этим элементам данных могут иметь доступ только четыре функции, прототипы которых появляются в определении класса (а также друзья класса). Обычно элементы данных перечисляются в разделе **private:** класса, а элементы-функции — в разделе **public:**. Как мы увидим далее, могут существовать элементы-функции с доступом **private** и данные с доступом **public:**.

Сразу после определения класса он может быть использован в качестве типа в объявлениях, подобных следующим:

```
Time sunset,                  // объект типа Time
     arrayOfTimes[5],         // массив объектов типа Time
     *poiterToTime,           // указатель на объект типа Time
     &dinnerTime = sunset;    // ссылка на объект типа Time
```

Имя класса становится новым спецификатором типа. Может существовать много объектов класса, подобно тому, как может быть много переменных типа

int. Программист может по мере необходимости создавать новые типы классов. Это одна из многих причин, благодаря которым C++ является *расширяемым языком*.

В программе на рис. 16.3 использован класс **Time**. Программа создает один экземпляр объекта класса **Time** с именем **t**. При создании экземпляра объекта автоматически вызывается конструктор **Time**, который явно инициализирует каждый закрытый элемент данных значением **0**. Затем для подтверждения того, что элементы данных были инициализированы должным образом, время выводится в военном и стандартном форматах. После этого время устанавливается при помощи элемента-функции **setTime** и снова выводится в обоих форматах. Потом элемент-функция **Time** пытается присвоить элементам данных недопустимые значения и время снова выводится в обоих форматах.

```
// FIG16_3.CPP
// Класс Time.
#include <iostream.h>

// Определение абстрактного типа данных (ADT)
class Time {
public:
   Time();                          // конструктор по умолчанию
   void setTime(int, int, int); // устанавливает элементы
                                    // hour, minute и second
   void printMilitary();    // выводит время в военном формате
   void printStandard();    // выводит время в стандартном формате

private:
   int hour;       // 0 - 23
   int minute;     // 0 - 59
   int second;     // 0 - 59
};

// Конструктор Time инициализирует каждый элемент данных нулем.
// Гарантирует, что все объекты Time изначально находятся
// в корректном состоянии.
Time::Time() { hour = minute = second = 0; }

// Устанавливает новое значение времени Time в военном формате.
// Проводит проверку правильности данных. Устанавливает
// недопустимые значения в нуль.
void Time::setTime( int h, int m, int s)
{
   hour = (h >= 0 && h < 24 ) ? h : 0;
   minute = (m >= 0 && m < 60 ) ? m : 0;
   second = (s >= 0 && s < 60 ) ? s : 0;
}

// Выводит Time в военном формате
void Time::printMilitary()
{
   cout << (hour < 10 ? "0" : "" ) << hour << ":"
        << (minute < 10 ? "0" : "" ) << minute << ":"
        << (srcond < 10 ? "0" : "" ) << second;
}
```

Рис. 16.3. Реализация абстрактного типа **Time** в виде класса (часть 1 из 2)

```
// Выводит время в стандартном формате
void Time::printStandard()
{
   cout << ((hour == 0 || hour == 12 ) ? 12 : hour % 12)
        << ":" << (minute < 10 ? "0" : "" ) << minute
        << ":" << (second < 10 ? "0" : "" ) << second
        << (hour < 12 ? " AM" : " PM");
}

// Программа для тестирования простого класса Time
main()
{
   Time t;  // создает экземпляр объекта t класса Time

   cout << "The initial military time is ";
   t.printMilitary();
   cout << "\nThe initial standard time is ";
   t.printStandard();

   t.setTime(13, 27, 6);
   cout << "\n\nMilitary time after setTime is ";
   t.printMilitary();
   cout << "\nStandard time after setTime is ";
   t.printStandard();

   t.setTime(99, 99, 99);  // присваивание недопустимых значений
   cout << "\n\nAfter attempting invalid settings :\n"
        << "Military time: ";
   t.printMilitary();
   cout << "\nStandard time: ";
   t.prindStandard();
   cout << endl;
   return 0;
}
```

```
The initial military time is 00:00:00

The initial standard time is 12:00:00 AM
Military time after setTime is 13:27:06

Standard time after setTime is 1:27:06 PM
After attempting invalid settings:
Military time: 00:00:00
Standard time: 12:00:00 AM
```

Рис. 16.3. Реализация абстрактного типа **Time** в виде класса (часть 2 из 2)

Еще раз обратите внимание, что элементам данных **hour**, **minute** и **second** предшествует метка **private:**. Обычно закрытые элементы данных класса недоступны за пределами этого класса. (Как уже говорилось, в главе 17 мы увидим, что к закрытым элементам данных класса могут обращаться друзья класса.) Идея состоит в том, что используемое внутри класса реальное представление данных не должно интересовать его клиентов. Именно в этом смысле говорят, что реализация класса *скрыта* от его пользователей. Такое *сокрытие информации* способствует модифицируемости программ и упрощает восприятие класса пользователем.

Общее методическое замечание 16.2

Клиенты класса использует этот класс, не зная внутренних деталей его реализации. При изменении реализации класса (например, из соображений эффективности) клиентов класса изменять не требуется. Это значительно упрощает модификацию программных систем.

В рассматриваемой нами программе конструктор класса **Time** просто инициализирует элементы данных значением 0 (т.е. военным эквивалентом времени 12 AM). Это гарантирует, что объект после его создания находится в корректном состоянии. Элементам данных не могут быть присвоены недействительные значения, поскольку при создании объекта **Time** конструктор вызывается автоматически, а все последующие попытки изменения элементов данных отслеживаются функцией **setTime**.

Общее методическое замечание 16.3

Обычно элементы-функции состоят всего лишь из нескольких строк кода, поскольку не требуется никакой логики для проверки корректности значений элементов данных.

Заметьте, что элементы данных не могут быть инициализированы при их объявлении в теле класса. Эти элементы данных либо должны быть инициализированы конструктором, либо им могут быть присвоены значения специальными функциями установки значений («set»-функциями).

Распространенная ошибка программирования 16.2

Попытка явной инициализации элемента данных класса.

Функция с тем же именем, что и класс, с предшествующим *символом тильды* (~), называется *деструктором* класса. Деструктор производит заключительную «приборку» каждого объекта класса перед тем, как выделенная для него память будет возвращена системе. Рассматриваемый пример не имеет деструктора. Мы обсудим конструкторы и деструкторы более подробно далее в этой главе и в главе 17.

Обратите внимание, что функциям, которые класс предусматривает для внешнего мира, предшествует метка **public:**. В открытых (публичных) функциях реализованы возможности, которые класс предоставляет своим клиентам. Открытые функции класса называют *интерфейсом* либо *открытым интерфейсом* класса.

Общее методическое замечание 16.4

Клиенты класса имеет доступ к интерфейсу класса, но они не должны иметь доступа к его реализации.

Определение класса содержит объявления его элементов данных и элементов-функций. Объявления элементов-функций представляют собой прототипы функций, которые мы обсуждали в предыдущих главах. Элементы-функции могут быть определены внутри класса, однако хорошим стилем является определение функций вне определения класса.

Хороший стиль программирования 16.2

Определяйте все элементы-функции, кроме самых маленьких, вне определения класса. Это способствует отделению интерфейса класса от его реализации.

Обратите внимание на использование *двухместной операции разрешения области действия* (::) в определении каждой из элементов-функций, следующих за определением класса на рис. 16.3. Сразу после определения класса и объявления его элементов-функций эти элементы-функции должны быть определены. Любая функция-элемент класса может быть определена непосредственно в его теле (вместо включения туда ее прототипа) или же она может быть определена после тела класса. Когда элемент-функция определяется после соответствующего определения класса, имени функции предшествуют имя класса и двухместная операция разрешения области действия (::). Поскольку различные классы могут иметь одинаковые имена элементов, операция разрешения области действия «привязывает» имя элемента к имени класса, однозначно идентифицируя функции-элементы данного класса.

Хотя функция-элемент, объявленная в определении некоторого класса, может быть определена вне его определения, эта функция, тем не менее, находится в *области действия класса*, т.е. ее имя известно только другим элементам данного класса, за исключением случаев, когда обращение к ней происходит через объект класса, ссылку на объект класса или указатель на объект класса. Вскоре мы поговорим подробнее об области действия класса.

Можно определять элементы-функции и в теле определения класса. Функции размером более одной или двух строк обычно определяются вне тела определения класса — это помогает отделить интерфейс класса от его реализации. Если функция-элемент определена в определении класса, то она автоматически становится встроенной. Элементы-функции, определенные вне определения класса, можно сделать встроенными путем явного использования ключевого слова **inline**. Не забывайте, что компилятор оставляет за собой право не делать встроенной любую функцию.

Общее методическое замечание 16.5

Объявление элементов-функций внутри определения класса с последующим определением их вне определения этого класса отделяет интерфейс класса от его реализации. Это способствует систематизации разработки программного обеспечения.

Совет по повышению эффективности 16.3

Определение элемента-функции небольшого размера внутри определения класса автоматически делает эту функцию встроенной (если с этим согласен компилятор). Это может повысить производительность системы, однако не всегда согласуется с принципами систематической разработки программного обеспечения.

Интересно отметить, что элементы-функции **printMilitary** и **printStandard** не принимают параметров. Это происходит потому, что элементы-функции косвенным образом знают, что они должны выводить элементы данных того конкретного объекта **Time**, для которого они были вызваны. Это делает вызовы элементов-функций более краткими в сравнении с традиционными вызовами в процедурном программировании.

Общее методическое замечание 16.6

Объектно-ориентированный подход к программированию часто приводит к упрощению вызовов функций из-за уменьшения числа передаваемых параметров. Это преимущество объектно-ориентированного программирования является следствием того, что инкапсуляция элементов данных и элементов-функций внутри объекта дает элементам-функциям право обращаться к этим элементам данных.

Классы упрощают программирование, поскольку клиент (или пользователь объекта класса) должен иметь дело только с инкапсулированными, т. е. встроенными в объект, действиями. При разработке такого рода действий обычно ориентируются на клиентов класса, а не на реализацию. Клиенты не должны иметь дела с реализацией класса. Интерфейсы *тоже изменяются, но* гораздо реже, чем реализации. При изменении реализации соответствующим образом должен измениться зависящий от реализации код. Скрывая реализацию, мы устраняем возможность зависимости других частей программы от деталей реализации класса.

Часто нет необходимости создавать классы «с нуля». Напротив, классы могут *производиться* от других классов, предоставляющих программисту действия, которые могут использовать новые классы. Классы также могут включать в качестве элементов объекты других классов. Такого рода *повторное использование программного обеспечения* может значительно повысить производительность труда программиста. Создание производных классов на основе существующих называется *наследованием* и подробно обсуждается в главе 19. Включение классов в качестве элементов других классов называется *композицией* и обсуждается в главе 17.

16.6. Область действия класса и доступ к элементам класса

Имена переменных и функций, объявленных в определении класса, а также элементы данных класса и его элементы-функции, принадлежат *области действия класса*. Функции, не являющиеся элементами класса, определяются в *области действия файла*.

Внутри области действия класса элементы класса непосредственно доступны для всех элементов-функций этого класса и допускают обращение просто по имени. Вне области действия класса на элементы класса можно ссылаться либо через имя объекта, либо через ссылку, либо через указатель на объект.

Элементы-функции класса могут быть перегружены, но только функциями в области действия этого класса. Для того, чтобы перегрузить элемент-функцию, просто задайте в области определения класса прототип для каждой версии перегруженной функции и напишите для каждой версии отдельное определение.

Распространенная ошибка программирования 16.3

Попытка перегрузить элемент-функцию класса функцией, не находящейся в области действия этого класса.

Элементы-функции имеют внутри класса *область действия функции*. Если в функции-элементе определена переменная с тем же именем, что и переменная с областью действия класса, то последняя будет скрыта переменной функции-элемента внутри области действия функции. Доступ к таким скрытым переменным можно получить посредством операции разрешения области действия, поместив перед этой операцией имя класса. Доступ к глобальным переменным может быть получен с помощью одноместной операции разрешения области действия (см. главу 15).

Операции, используемые для доступа к элементам класса, идентичны операциям для доступа к элементам структуры. Для доступа к элементам объекта применяется операция выбора элемента — точка (.) — в сочетании с именем объекта или со ссылкой на объект. Операция выбора элемента — стрелка (->) — применяется в сочетании с указателем на объект.

В программе на рис. 16.4 для иллюстрации доступа к элементам класса при помощи операций выбора элементов используется простой класс с именем **Count**, который содержит открытый элемент данных **x** типа **int** и открытую функцию-элемент **print**. Программа создает экземпляры трех переменных типа **Count** — **counter**, **counterRef** (ссылка на объект типа **Count**) и **counterPtr** (указатель на объект типа **Count**). Переменная **counterRef** объявляется в качестве ссылки на переменную **counter**, а переменная **counterPtr** объявляется как указатель на **counter**. Важно обратить внимание на то, что здесь элемент данных **x** был сделан открытым просто для демонстрации того, каким образом происходит обращение к открытым элементам. Как мы констатировали, данные обычно делают закрытыми; во всех последующих примерах мы будем поступать именно так.

16.7. Отделение интерфейса от реализации

Одним из наиболее фундаментальных принципов систематической разработки программного обеспечения является отделение интерфейса от реализации. Это упрощает модификацию программ. Что касается клиентов класса, изменения в реализации класса не затрагивают их до тех пор, пока не будет изменен его интерфейс.

Общее методическое замечание 16.7

Помещайте объявление класса в заголовочный файл, который должен быть включен любым клиентом, желающим использовать этот класс. Это — открытый интерфейс класса. Помещайте определения элементов-функций этого класса в отдельный исходный файл. Это — реализация класса.

Общее методическое замечание 16.8

Клиентам класса не обязательно видеть его исходный код для того, чтобы пользоваться этим классом. Однако клиентам необходимо иметь возможность компоноваться с объектным кодом класса.

Это поощряет поставку независимыми производителями программного обеспечения библиотек классов для продажи или лицензирования. Независимые производители поставляют свои программные продукты, включая в них

только заголовочные файлы и объектные модули. При этом не раскрывается какой-либо информации, связанной с авторским правом, — как это было бы в случае предоставления исходного кода. Сообщество пользователей C++ выигрывает, таким образом, от большего числа доступных библиотек классов, произведенных независимыми фирмами.

```
// FIG16_4.CPP
// Демонстрация операций доступа к элементам класса . и ->
//
// Внимание: в дальнейших примерах мы будем использовать закрытые
// данные.
#include <iostream.h>

// Простой класс Count
class Count {
public:
   int x;
   void print ()   { cout << x << '\n' ; }
} ;

main ()
{
   Count counter,           // создает объект counter
   *counterPtr = &counter, // указатель на counter
   &counterRef = counter;  // ссылка на counter

   cout << "Assign 7 to x and print using the object's name: ";
   counter.x = 7;          // присваивает 7 элементу данных x
   counter.print();        // вызывает элемент-функцию print

   cout << "Assign 8 to x and print using a reference: ";
   counterRef.x = 8;       // присваивает 8 элементу данных x
   counterRef.print ();    // вызывает элемент-функцию print

   cout << "Assign 10 to x and print using a pointer: ";
   counterRtr->x = 10;     // присваивает 10 элементу данных x
   counterRtr->print();    // вызывает элемент-функцию print
   return 0;
}
```

```
Assign 7 to x and print using the object's name: 7
Assign 8 to x and print using a reference: 8
Assign 10 to x and print using a pointer: 10
```

Рис. 16.4. Доступ к элементам данных и элементам-функциям объекта через имя объекта, ссылку и указатель на объект

Однако в реальности дело обстоит не так хорошо, как могло бы показаться. В заголовочных файлах все же содержится некоторая часть реализации и скрытых намеков на нее. Например, встроенные элементы-функции обязаны находиться в заголовочном файле, чтобы при компиляции кода клиента компилятор мог включать в него в нужных местах определение встроенной функции. Закрытые элементы класса перечислены в его объявлении в заголовочном файле, поэтому эти элементы видимы для клиентов, даже если клиенты могут и не иметь к ним доступа.

Общее методическое замечание 16.9

Информация, являющаяся важной для интерфейса класса, должна включаться в заголовочный файл. Информация, которая будет использоваться только внутри класса и не потребуется клиентам, должна находиться в неопубликованном исходном файле. Это еще один пример принципа минимума привилегий.

На рис. 16.5 показана программа из рис. 16.3, разбитая на несколько файлов. При построении программы на C++ каждое определение класса обычно помещается в *заголовочный файл*, а определения элементов-функций этого класса помещаются в *файлы исходного кода* с тем же базовым именем. Заголовочные файлы включаются (посредством **#include**) во все файлы, использующие этот класс, а исходный файл с определениями элементов-функций компилируется и компонуется с файлом, содержащим главную программу. См. документацию по вашему компилятору, чтобы выяснить, каким образом компилировать и компоновать программы, состоящие из нескольких исходных файлов.

```
// TIME1.H
// Объявление класса Time.
// Элементы-функции определены в TIME1.CPP

// не допускает повторных включений заголовочного файла
#ifndef TIME1_H
#define TIME1_H

// Определение абстрактного типа данных Time
class Time  {
public:
   Time();                          // конструктор по умолчанию
   void setTime(int, int, int); // устанавливает элементы
                                    // hour, minute и second
   void printMilitary();    // выводит время в военном формате
   void printStandard();    // выводит время в стандартном формате
private:
   int hour;      // 0 - 23
   int minute;    // 0 - 59
   int second;    // 0 - 59
};

#endif
```

Рис. 16.5. Заголовочный файл класса **Time** (часть 1 из 3)

Программа на рис. 16.5 состоит из заголовочного файла **time1.h**, в котором объявлен класс **Time**, файла **time1.cpp**, в котором определены элементы-функции класса **Time**, и файла **fig16_5.cpp**, в котором определена функция **main**. Выходные данные этой программы идентичны выходным данным программы на рис. 16.3.

Обратите внимание, что объявление класса окружено следующим препроцессорным кодом (см. главу 13):

```
// не допускает многократных включений заголовочного файла
#ifndef TIME1_H
#define TIME1_H
   ...
#endif
```

При создании программ большего размера в заголовочные файлы будут также помещены и другие определения и объявления. Предыдущие директивы препроцессора не допускают включения кода, находящегося между **#ifndef** и **#endif**, если было определено имя **TIME1_H**. Если заголовок ранее не был включен в файл, директива **#define** определяет имя **TIME1_H** и происходит включение операторов заголовочного файла. Если заголовок был включен ранее, то имя **TIME1_H** уже определено и заголовочный файл повторно не включается. Замечание: по соглашению, которое мы используем для имен символических констант в директивах препроцессора, имя такой константы представляет собой просто имя заголовочного файла с символом подчеркивания вместо точки.

```
// TIME1.CPP
// Определения элементов-функций класса Time.
#include <iostream.h>
#include "time1.h"

// Конструктор Time инициализирует каждый элемент данных нулем.
// Гарантирует, что все объекты Time изначально находятся
// в непротиворечивом состоянии.
Time::Time() { hour = minute = second = 0; }

// Устанавливает новое значение времени Time в военном формате.
// Проводит проверку правильности данных.
// Устанавливает недопустимые значения в нуль
// (непротиворечивое состояние).
void Time::setTime( int h, int m, int s)
{
   hour = (h >= 0 && h < 24 ) ? h : 0;
   minute = (m >= 0 && m < 60 ) ? m : 0;
   second = (s >= 0 && s < 60 ) ? s : 0;
}

// Выводит Time в военном формате
void Time::printMilitary()
{
   cout << (hour < 10 ? "0" : "" ) << hour << ":"
        << (minute < 10 ? "0" : "" ) << minute << ":"
        << (srcond < 10 ? "0" : "" ) << second;
}

// Выводит время в стандартном формате
void Time::printStandard()
{
   cout << ((hour == 0 || hour == 12 ) ? 12 : hour % 12)
        << ":" << (minute < 10 ? "0" : "" ) << minute
        << ":" << (second < 10 ? "0" : "" ) << second
        << (hour < 12 ? " AM" : " PM");
}
```

Рис. 16.5. Исходный файл определений элементов-функций класса **Time** (часть 2 из 3)

Хороший стиль программирования 16.3

Используйте директивы препроцессора **#ifndef**, **#define** и **#endif**, чтобы не допустить повторного включения в программу заголовочных файлов.

```
// FIG16_5.CPP
// Программа-тестер для класса Time1
// ЗАМЕЧАНИЕ: Компилировать с TIME1.CPP
#include <iostream.h>
#include "time1.h"

// Программа для тестирования простого класса Time
main()
{
   Time t;  // создает экземпляр объекта t класса Time

   cout << "The initial military time is ";
   t.printMilitary();
   cout << "\nThe initial standard time is ";
   t.printStandard();

   t.setTime(13, 27, 6);
   cout << "\n\nMilitary time after setTime is ";
   t.printMilitary();
   cout << "\nStandard time after setTime is ";
   t.printStandard();

   t.setTime(99, 99, 99);  // присваивание недопустимых значений
   cout << "\n\nAfter attempting invalid settings :\n"
        << "Military time: ";
   t.printMilitary();
   cout << "\nStandard time: ";
   t.prindStandard();
   cout << endl;
   return 0;
}
```

```
The initial military time is 00:00:00
The initial standard time is 12:00:00 AM

Military time after setTime is 13:27:06
Standard time after setTime is 1:27:06 PM

After attempting invalid settings:
Military time: 00:00:00
Standard time: 12:00:00 AM
```

Рис. 16.5. Программа-тестер для класса **Time** (часть 3 из 3)

Хороший стиль программирования 16.4

Используйте имя заголовочного файла с подчеркиванием вместо точки в директивах препроцессора **#ifndef** и **#define** заголовочных файлов.

16.8. Управление доступом к элементам класса

Метки **public:** и **private:** (и **protected:**, как мы увидим в главе 19, «Наследование») используются для управления доступом к элементам данных и элементам-функциям класса. Режимом доступа по умолчанию для классов является **private:**, поэтому все элементы после заголовка класса и до первой метки

являются закрытыми. После каждой метки задаваемый ею режим действует вплоть до следующей метки или до завершающей правой фигурной скобки (}) определения класса. Метки **public:**, **private:** и **protected:** могут повторяться, однако такое их использование нечасто и может приводить к путанице.

К закрытым элементам класса могут иметь доступ только элементы (и друзья) этого класса. К открытым элементам класса можно обращаться из любой функции программы.

Основным назначением открытых элементов класса является предоставление клиентам информации о доступных в классе *услугах*. Этот набор услуг образует *открытый интерфейс* класса. Клиенты класса не должны заботиться о способе выполнения классом своих задач. Закрытые элементы класса, как и определения его открытых элементов-функций, недоступны для клиентов класса. Эти компоненты составляют реализацию класса.

Общее методическое замечание 16.10

C++ поощряет независимость программ от реализации. Когда реализация класса, который используется в независимом от реализации коде, изменяется, этот код не нуждается в модификации, однако может потребоваться его перекомпиляция.

Распространенная ошибка программирования 16.4

Попытка функции, не являющейся элементом данного класса (или его другом), получить доступ к его закрытому элементу.

Рис. 16.6 демонстрирует, что закрытые элементы класса доступны только через открытый интерфейс этого класса посредством вызова открытых элементов-функций. Во время компиляции этой программы компилятор генерирует две ошибки, сообщая, что закрытые элементы, на которые ссылаются операторы, является недоступным. Программа на рис. 16.6 включает **time1.h** и компилируется с файлом **time1.cpp**, приведенным на рис. 16.5.

Хороший стиль программирования 16.5

Если вы хотите сначала перечислить закрытые элементы в объявлении класса, явно используйте метку **private:**, несмотря на то, что тип доступа **private:** принят по умолчанию. Это делает программу более понятной. Мы предпочитаем сначала перечислять элементы класса с типом доступа **public:**.

Хороший стиль программирования 16.6

Несмотря на то, что метки **public:** и **private:** могут повторяться и чередоваться, перечислите сначала в одной группе все открытые элементы класса, а затем перечислите в другой группе все его закрытые элементы. Это акцентирует внимание пользователя на открытом интерфейсе класса, а не на его реализации.

Общее методическое замечание 16.11

Оставляйте все элементы данных класса закрытыми. Предусмотрите публичные элементы-функции для установки и получения значений закрытых элементов данных. Такая архитектура помогает скрывать реализацию класса от его клиентов, что уменьшает количество ошибок и способствует модифицируемости программы.

```
// FIG16_6.CPP
// Демонстрирует ошибки, возникающие в результате попыток
// получить доступ к закрытым элементам класса.
#include <iostream.h>
#include "time1.h"

main()
{
   Time t;

   // Ошибка: элемент 'Time::hour' недоступен
   t.hour = 7;

   // Ошибка: элемент 'Time::minute' недоступен
   cout << "minute = " << t.minute;

   return 0;
}
```

```
Compiling FIG16_6.CPP:
Error FIG16_6.CPP: 12: 'Time::hour' is not accessible
Error FIG16_6.CPP: 15: 'Time::minute' is not accessible
```

Рис. 16.6. Вызвавшая ошибку попытка обращения к закрытым элементам класса

Клиентом класса может быть как функция-элемент другого класса, так и глобальная функция.

Доступом по умолчанию для элементов класса является **private**. Доступ к элементам класса может быть явно сделан защищенным или открытым. Доступом по умолчанию для элементов структур и объединений является **public**. Доступ к элементам структуры может быть явно сделан открытым или закрытым (или, как мы увидим в главе 19, защищенным). С объединением спецификаторы доступа к элементам явно использоваться не могут.

Общее методическое замечание 16.12

Проектировщики классов используют закрытые, защищенные и открытые элементы для реализации принципов сокрытия информации и минимальных привилегий.

Обратите внимание, что элементы класса являются закрытыми по умолчанию, поэтому всегда можно обойтись без явного использования спецификатора доступа к элементам **private:**. Однако многие программисты предпочитают сначала перечислять интерфейс класса (т.е. открытые элементы класса); затем перечисляются закрытые элементы класса и, таким образом, возникает потребность в явном использовании спецификатора доступа **private:** в определении класса.

Хороший стиль программирования 16.7

Использование каждого из спецификаторов доступа к элементам **public:**, **protected:** и **private:** только один раз в определении класса помогает избежать путаницы.

То, что данные класса являются закрытыми, не обязательно означает, что клиенты не могут производить изменений в этих данных. Данные могут быть

изменены элементами-функциями или друзьями класса. Как мы увидим, такие функции должны разрабатываться так, чтобы гарантировать целостность данных.

Доступ к закрытым данным класса может строго контролироваться путем использования элементов-функций, называемых *функциями доступа*. Например, для того, чтобы разрешить клиентам считывать значение закрытого элемента данных, класс может предусматривать так называемую *«get»-функцию*. Чтобы давать возможность клиентам изменять закрытые данные, класс может иметь так называемую *«set»-функцию*. Может показаться, что такого рода изменение данных нарушает представление о закрытых данных. Однако set-функция может иметь средства проверки допустимости данных (например, проверку диапазона), чтобы гарантировать правильную установку соответствующего значения. Get-функция не обязана выдавать данные в «сыром» виде; напротив, get-функция может редактировать данные и ограничивать представление данных, которые будет видеть клиент.

Общее методическое замечание 16.13

Задание закрытого доступа для элементов данных класса и открытого доступа для его элементов-функций облегчает отладку, поскольку проблемы, связанные с манипулированием данными, ограничиваются либо элементами-функциями класса, либо его друзьями.

16.9. Функции доступа и сервисные функции

Не все элементы-функции целесообразно делать общедоступными, т.е. частью интерфейса класса. Некоторые элементы-функции остаются закрытыми и служат в качестве вспомогательных для других функций класса.

Общее методическое замечание 16.14

Элементы-функции имеют тенденцию подразделяться на ряд различных категорий: функции, которые считывают и возвращают значения закрытых элементов данных, функции, которые устанавливают значения закрытых элементов данных, функции, которых реализуют характерные особенности класса, и функции, которые выполняют разнообразную механическую работу для класса, такую, как инициализация объектов класса, присваивание объектов, преобразование между классами и встроенными типами данных или между классами и другими классами, и управление памятью для объектов класса.

Функции доступа могут считывать или отображать данные. Другим распространенным применением функций доступа является проверка истинности либо ложности условий — подобные функции часто называют *предикатными функциями*. Примером предикатной функции может служить функция **isEmpty** для любого контейнерного класса, такого, как связанный список, стек или очередь. Программа обычно проверяет условие **isEmpty** перед попыткой считывания еще одного элемента из контейнерного объекта. Предикатная функция **isFull** может проверять объект контейнерного класса, чтобы установить, что в нем отсутствует дополнительное пространство.

На рис. 16.7 демонстрируется понятие *сервисной функции*. Сервисная функция не является частью интерфейса класса. Это закрытая функция-эле-

мент, которая поддерживает работу открытых функций-элементов класса. Сервисные функции не предназначены для использования клиентами класса.

```
// SALESP.H
// Определение класса SalesPerson
// Элементы-функции определены в SALESP.CPP

#ifndef SALESP_H
#define SALESP_H

class SalesPerson {
public:
   SalesPerson();          // конструктор
   void setSales();        // пользователь задает показатели сбыта
   void printAnnualSales();

private:
   double sales [13];          // 12 ежемесячных показателей сбыта
   double totalAnnualSales(); // сервисная функция
};

#endif

// SALESP.CPP
// Элементы-функции для класса SalesPerson
#include <iostream.h>
#include <iomanip.h>
#include "salesp.h"

// Конструктор инициализирует массив
SalesPerson::SalesPerson()
{
   for (int i = 0; i <= 12; i++)
      sales[i] = 0.0;
}

// Функция для установки 12 ежемесячных показателей сбыта
void SalesPerson::setSales()
{
   for (int i = 1; i <= 12; i++) {
      cout << "Enter sales amount for month "
           << i << ":" ;
      cin >> sales [i];
   }
}

// Закрытая сервисная функция для вычисления годового сбыта
double SalesPerson::totalAnnualSales()
{
   double total = 0.0;

   for (int i = 1; i <= 12; i++)
      total += sales[i];

   return total;
}
```

Рис. 16.7. Использование сервисной функции (часть 1 из 2)

```
// Выводит итоговый годовой сбыт
void SalesPerson::printAnnualSales()
{
   cout << setprecision (2)
        << setiosflags (ios::fixed | ios::showpoint)
        << "\nThe total annual sales are: $"
        << totalAnnualSales() << endl;
}

// FIG16_7.CPP
// Демонстрация сервисной функции
// Компилируется с SALESP.CPP
#include "salesp.h"

main()
{
   SalesPerson s;          // создает объект s класса SalesPerson

   s.setSales();
   s.printAnnualSales();

   return 0;
}
```

```
Enter sales amount for month 1:  5314.76
Enter sales amount for month 2:  4292.38
Enter sales amount for month 3:  4589.83
Enter sales amount for month 4:  5534.03
Enter sales amount for month 5:  4376.34
Enter sales amount for month 6:  5698.45
Enter sales amount for month 7:  4439.22
Enter sales amount for month 8:  5893.57
Enter sales amount for month 9:  4909.67
Enter sales amount for month 10:  5123.45
Enter sales amount for month 11:  4024.97
Enter sales amount for month 12:  5923.92

The total annual sales are: $60120.59
```

Рис. 16.7. Использование сервисной функции (часть 2 из 2)

Класс **SalesPerson** содержит массив из 12 ежемесячных показателей сбыта, инициализируемых нулями в конструкторе; функция **setSales** присваивает показателям сбыта задаваемые пользователем значения. Открытая функция-элемент **printAnnualSales** выводит итоговый сбыт за последние 12 месяцев. Сервисная функция **totalAnnualSales** суммирует 12 ежемесячных показателей сбыта для функции **printAnnualSales**. Элемент-функция **printAnnualSales** преобразует показатели сбыта в долларовый формат. Каждая из используемых здесь возможностей форматирования языка C++ более подробно объясняется в главе 21. Сейчас мы дадим только краткое пояснение. Вызов

```
setprecision(2)
```

указывает, что справа от десятичной точки должны выводиться два знака «точности», как нам и требуется для долларовых сумм вроде 23.47. Этот вызов называется «параметризованным манипулятором потока». Программа,

применяющая такого рода вызовы, должна также содержать директиву препроцессора

```
#include <iomanip.h>
```

Строка

```
setiosflags(ios::fixed | ios::showpoint);
```

задает отображение показателя сбыта в так называемом формате с фиксированной точкой (в противоположность экспоненциальному формату). Опция **showpoint** принудительно выводит десятичную точку и нули в конце числа, даже если последнее является целой долларовой суммой, как в случае 47.00. В C++ такое число было бы выведено просто как 47, если бы опция **showpoint** не была установлена.

16.10. Инициализация объектов класса: конструкторы

После создания объектов их элементы могут быть инициализированы с помощью конструкторов. Конструктор представляет собой функцию-элемент класса с тем же именем, что и класс. Программист пишет конструктор, который затем автоматически вызывается всякий раз, когда создается объект этого класса. *Элементы данных не могут быть инициализированы в определении класса.* Элементы данных либо должны инициализироваться в конструкторе класса, либо их значения могут быть установлены позже, после создания объекта. Конструктор не может ни специфицировать тип возвращаемого значения, ни возвращать какое-либо значение. Конструкторы могут быть перегружены, чтобы предусмотреть различные способы инициализации объектов класса.

Распространенная ошибка программирования 16.5

Попытка явной инициализации элемента данных в определении класса.

Распространенная ошибка программирования 16.6

Попытка объявить для конструктора тип возвращаемого значения и/или попытка возвратить значение из конструктора.

Хороший стиль программирования 16.8

Когда это целесообразно (почти всегда), определяйте конструктор, чтобы гарантировать надлежащую инициализацию каждого объекта осмысленными значениями.

Хороший стиль программирования 16.9

Любая функция-элемент (или дружественная функция), изменяющая закрытые элементы данных объекта, должна гарантировать, что данные остаются в целостном (действительном) состоянии.

При объявлении объекта класса справа от его имени и до точки с запятой в круглых скобках могут быть заданы *инициализаторы*. Эти инициализаторы передаются в качестве аргументов в конструктор класса. Скоро мы рассмотрим несколько примеров таких *вызовов конструктора*.

16.11. Использование с конструкторами аргументов по умолчанию

Конструктор из листинга **time1.cpp** (рис. 16.5) инициализировал **hour**, **minute** и **second** нулевыми значениями (т.е. 12 часами полуночи в военном формате времени). Конструкторы могут содержать аргументы по умолчанию. На рис. 16.8 конструктор класса **Time** переопределяется так, чтобы он включал для каждой переменной аргумент с нулевым значением по умолчанию. Задавая для конструктора аргумент по умолчанию, мы гарантируем, что благодаря этим аргументам объект будет находится в корректном состоянии, даже если не будет задано никаких значений при вызове конструктора. Предусмотренный программистом конструктор, все аргументы которого имеют значения по умолчанию, называется также конструктором по умолчанию (он может вызываться без аргументов). Конструктор использует того же рода код проверки допустимости данных, что и функция **setTime**, чтобы гарантировать, что значение, заданное для элемента **hour**, находится в диапазоне от 0 до 23 и что значения для каждого из элементов **minute** и **second** находятся в диапазоне от 0 до 59. Если значение попадает за пределы диапазона, оно устанавливается в нуль (это — пример обеспечения целостности состояния данных объекта). Конструктор мог бы непосредственно вызывать **setTime**, однако это привело бы к накладным расходам, связанным с дополнительным вызовом функции. В программе на рис. 16.8 инициализируются пять объектов класса **Time** — один со всеми аргументами, принимаемыми по умолчанию при вызове конструктора, один с одним указанным аргументом, один с двумя указанными аргументами, один с тремя указанными аргументами и один с тремя недопустимыми значениями аргументов. Содержание элементов данных каждого объекта после его создания и инициализации отображается на экране.

```
// TIME2.H
// Объявление класса Time.
// Элементы-функции определены в TIME2.CPP

// не допускает повторных включений заголовочного файла
#ifndef TIME2_H
#define TIME2_H

class Time {
public:
   Time(int = 0; int = 0; int = 0);  // конструктор по умолчанию
   void setTime(int, int, int);
   void printMilitary();
   void printStandard();
private:
   int hour;
   int minute;
   int second;
};

#endif
```

Рис. 16.8. Использование конструктора с аргументами по умолчанию (часть 1 из 3)

```
// TIME2.CPP
// Определения элементов-функций для класса Time.
#include "time2.h"
#include <iostream.h>

// Конструктор для инициализации закрытых данных.
// Значения по умолчанию равны 0 (см. определение класса).
Time::Time(int hr, int min, int sec)
{
   hour = ( hr >= 0 && hr < 24 ) ? hr : 0;
   minute = ( min >= 0 && min < 60 ) ? min : 0;
   second = ( sec >= 0 && sec < 60 ) ? sec : 0;
}

// Устанавливает значения hour, minute и second.
// Недопустимые знасения устанавливаются в 0.
void Time::setTime(int h, int m, int s)
{
   hour = ( h >= 0 && h < 24) ? h : 0;
   minute = ( m >= 0 && m < 60) ? m : 0;
   second = ( s >= 0 && s < 60) ? s : 0;
}

// Отображает время в военном формате: HH:MM:SS
void Time::printMilitary()
{
   cout << (hour < 10 ? "0" : "") << hour << ":"
        << (minute < 10 ? "0" : "") << minute << ":"
        << (second < 10 ? "0" : "") << second;
}

// Отображает время в стандартном формате: HH:MM:SS AM (или PM)
void Time::printStandard()
{
   cout << ((hour == 0 || hour == 12) ? 12 : hour % 12)
        << ":" << (minute < 10 ? "0" : "") << minute
        << ":" << (second < 10 ? "0" : "") << second
        << (hour < 12 ? " AM" : " PM");
}

// FIG16_8.CPP
// Демонстрация конструктора по умолчанию
// для класса Time.
#include <iosrteam.h>
#include "time2.h"

main()
{
   Time t1, t2(2), t3(21, 34), t4(12, 25, 42),
      t5 (27, 74, 99);

   cout << "Consrtucted with:\n"
        << "all arguments defaulted: \n   ";
   t1.printMilitary();
```

Рис. 16.8. Использование конструктора с параметрами по умолчанию (часть 2 из 3)

```
   cout << "\n    ";
   t1.printStandard();

   cout << "\nhour specified; minute and second defaulted:\n    ";
   t2.printMilitary();
   cout << "\n    ";
   t2.printStandard();

   cout << "\nhour and minute specified; second defaulted:\n    ";
   t3.printMilitary();
   cout << "\n    ";
   t3.printStandard();

   cout << "\nhour, minute and second specified:\n    ";
   t4.printMilitary();
   cout << "\n    ";
   t4.printStandard();

   cout << "\nall invalid values specified:\n    ";
   t5.printMilitary();
   cout << "\n    ";
   t5.printStandard();
   cout << endl;

   return 0;
}
```

```
Constructed with:
all arguments defaulted:
   00:00:00
   12:00:00 AM
hour specified; minute and second defaulted:
   02:00:00
   2:00:00 AM
hour and minute specified; second defaulted:
   21:34:00
   9:34:00 PM
hour, minute and second specified:
   12:25:42
   12:25:42 PM
all invalid values specified:
   00:00:00
   12:00:00 AM
```

Рис. 16.8. Использование конструктора с параметрами по умолчанию (часть 3 из 3)

Если для класса не определено ни одного конструктора, компилятор создает конструктор по умолчанию. Такой конструктор не выполняет никакой инициализации, поэтому после создания объекта не гарантируется его корректное состояние.

Хороший стиль программирования 16.10

Всегда предусматривайте конструктор, который выполняет соответствующую инициализацию объектов своего класса.

16.12. Деструкторы

Деструктор — это специальная функция-элемент класса. Имя деструктора состоит из символа тильды (~), за которым следует имя класса. Такое соглашение о наименовании не противоречит интуиции, поскольку операция тильды является поразрядной операцией дополнения, а в каком-то смысле деструктор является дополнением конструктора.

Деструктор класса вызывается автоматически, когда объект класса выходит из области действия. Сам деструктор фактически не разрушает объекта, скорее он выполняет заключительную «приборку», прежде чем системе будет возвращена выделенная объекту память.

Деструктор не принимает параметров и не возвращает значения. У класса может быть только один деструктор — перегрузка деструкторов не допускается.

Распространенная ошибка программирования 16.7

Попытка передать параметры в деструктор, возвратить значение из деструктора или перегрузить деструктор.

Обратите внимание, что мы не определяли деструкторов для представленных до сих пор классов. На самом деле деструкторы редко используются с простыми классами. В главе 18 мы увидим, что применение деструкторов целесообразно для классов, объекты которых содержат динамически выделенную память (например, для массивов и строк).

16.13. Когда вызываются конструкторы и деструкторы

Обычно конструкторы и деструкторы вызываются автоматически. Порядок, в котором происходят обращения к этим функциям, зависит от того порядка, в каком объекты входят в область действия и выходят из нее.

Как правило, вызовы деструкторов происходят в порядке, обратном порядку вызова конструкторов. Однако период хранения объектов может влиять на порядок вызова деструкторов.

Для объектов, объявленных в глобальной области действия, конструкторы вызываются в начале выполнения программы. Соответствующие деструкторы вызываются при ее завершении.

Для автоматических локальных объектов конструкторы вызываются при их объявлении. Соответствующие деструкторы вызываются, когда объекты выходят из области действия (т.е. происходит выход из блока, в котором они объявлены). Обратите внимание, что конструкторы и деструкторы для автоматических локальных объектов могут вызываться многократно по мере того, как объекты входят в область действия и выходят из нее.

Для статических локальных объектов конструкторы вызываются один раз при объявлении этих объектов. Соответствующие деструкторы вызываются при завершении работы программы.

Программа на рис. 16.9 демонстрирует порядок, в котором вызываются конструкторы и деструкторы для объектов типа **CreateAndDestroy**, находящихся в различных областях действия. Программа объявляет объект **first** в глобальной области действия. Его конструктор вызывается в начале выполнения программы, а деструктор — при завершении работы программы после разрушения всех других объектов.

```
// CREATE.H
// Определение класса CreateAndDestroy.
// Элементы-функции определены в CREATE.CPP.

#ifndef CREATE_H
#define CREATE_H

class CreateAndDestroy {
public:
   CreateAndDistroy(int);     // конструктор
   ~CreateAndDestroy();       // деструктор
private:
   int data;
};

#endif

// CREATE.CPP
// Определения элементов-функций для класса CreateAndDestroy
#include <iostream.h>
#include "create.h"

CreateAndDestroy::CreateAndDestroy(int value)
{
   data = value;
   cout << "Object " << data << " constructor";
}

CreateAndDestroy::~CreateAndDestroy()
   { cout << "Object " << data << "   destructor " << endl; }

// FIG16_9.CPP
// Демонстрация порядка, в котором вызываются
// конструкторы и деструкторы.
#include <iostream.h>
#include "create.h"

void create(void);   // прототип

CreateAndDestroy first(1);   // глобальный объект

main()
{
   cout << "  (global created before main) \n";

   CreateAndDestroy second(2);      // локальный объект
   cout << "  (local automatic in main) \n";
```

Рис. 16.9. Демонстрация порядка, в котором вызываются конструкторы и деструкторы (часть 1 из 2)

```
   static CreateAndDestroy third(3);  // локальный объект
   cout << "  (local static in main) \n";

   create();  // вызывает функцию, создающую объекты

   CreateAndDestroy fourth(4);     // локальный объект
   cout << "  (local automatic in main) \n";
   return 0;
}

// Функция, создающая объекты
void create(void)
{
   CreateAndDestroy fifth(5);
   cout << "  (local automatic in create) \n";

   static CreateAndDestroy sixth(6);
   cout << "  (local static in create) \n";

   CreateAndDestroy seventh(7);
   cout << "  (local automatic in create ) \n";
}
```

Рис. 16.9. Демонстрация порядка, в котором вызываются конструкторы и деструкторы (часть 2 из 3)

```
Object 1   constructor   (global created before main)
Object 2   constructor   (local automatic in main)
Object 3   constructor   (local static in main)
Object 5   constructor   (local automatic in create)
Object 6   constructor   (local static in create)
Object 7   constructor   (local automatic in create)
Object 7   destructor
Object 5   destructor
Object 4   constructor   (local automatic in main)
Object 4   destructor
Object 2   destructor
Object 6   destructor
Object 3   destructor
Object 1   destructor
```

Рис. 16.9. Демонстрация порядка, в котором вызываются конструкторы и деструкторы (часть 2 из 2)

В функции **main** объявлены три объекта. Объекты **second** и **fourth** являются локальными автоматическими объектами, а объект **third** статическим локальным объектом. Конструкторы для каждого из этих объектов вызываются при объявлении каждого из них. Деструкторы для объектов **fourth** и **second** вызываются (в таком порядке) при достижении конца функции **main**. Поскольку объект **third** является статическим, он существует с момента своего объявления до завершения работы программы. Деструктор для объекта **third** вызывается перед деструктором для объекта **first**, но после разрушения всех других объектов.

В функции create объявлены три объекта. Объекты **fifth** и **seventh** являются автоматическими локальными объектами, а объект **sixth** статическим локальным объектом. Деструкторы для объектов **seventh** и **fifth** вызываются

(в таком порядке) при достижении конца функции **create**. Поскольку объект **sixth** является статическим, он существует с момента своего объявления до завершения работы программы. Деструктор для объекта **sixth** вызывается перед деструкторами для объектов **third** и **first**, но после разрушения всех остальных объектов.

16.14. Использование элементов данных и элементов-функций

Закрытыми элементами данных могут манипулировать только элементы-функции (и друзья) класса. Типичным примером может служить корректировка банковского баланса клиента (т.е. закрытого элемента данных класса **BankAccount**) с помощью элемента-функции **computeInterest**.

Классы часто предусматривают открытые элементы-функции, чтобы дать возможность клиентам класса устанавливать (т. е. записывать) или получать (т.е. считывать) значения закрытых элементов данных. Эти функции не обязаны называться именно «set»- (установить) и «get»- (получить) функциями, но часто их называют именно так. Более конкретно, элемент-функция, устанавливающая элемент данных **interestRate**, обычно называется **setInterestRate**, а элемент-функция, которая получает значение **interestRate**, обычно называется **getInterestRate**. Get-функции часто называют также «функциями опроса».

Может показаться, что предоставить клиенту возможность как установки, так и получения значений элементов данных — это, по сути, то же самое, что сделать эти элементы открытыми. В этом состоит еще одна тонкость языка C++, которая делает его столь привлекательным для разработчиков программного обеспечения. Если элемент данных является открытым, то любая функция в программе может произвольно считывать этот элемент данных или записывать в него информацию. Если элемент данных является закрытым, то, конечно, может показаться, что общедоступная «get»-функция дает возможность другим функциям произвольно считывать данные, однако эта get-функция может контролировать форматирование и отображение данных. Общедоступная set-функция может — и, по всей вероятности, будет — тщательно контролировать любую попытку изменения значения элемента данных. Это гарантирует, что новое значение элемента данных будет соответствовать его смыслу. Например, будут отвергнуты попытка присвоить дню месяца значение 37, попытка присвоить весу человека отрицательное значение, попытка присвоить числовой величине литерное значение, попытка присвоить экзаменационной оценке значение 185 (при возможном диапазоне от нуля до 100) и т.д.

Общее методическое замечание 16.15

Закрытие элементов данных и контроль доступа (особенно доступа для записи) к этим элементам посредством открытых элементов-функций помогает гарантировать целостность данных.

Выгоды, связанные с целостностью данных, не реализуются автоматически просто потому, что элементы данных сделаны закрытыми, — программист должен обеспечить проверку допустимости данных. Однако C++ предоставляет программистам некую основу, создающую удобства при проектировании более качественных программ.

Хороший стиль программирования 16.11

Элементы-функции, присваивающие значения закрытым данным, должны проверять корректность вновь вводимых значений; в случае их некорректности set-функции должны приводить закрытые элементы в некоторое состояние, гарантирующее целостность данных.

В программе на рис. 16.10 производится расширение нашего класса **Time** путем включения get- и set-функций для закрытых элементов данных **hour**, **minute** и **second**. Set-функции строго контролируют установку элементов данных. Попытки присвоить любому элементу данных некорректное значение вызывают установку этого элемента данных в нуль (оставляя, таким образом, данные в целостном состоянии). Каждая get-функция просто возвращает значение соответствующего элемента данных. Сначала программа использует set-функции для присваивания закрытым элементам данных объекта **t** класса **Time** допустимых значений, затем она использует get-функции для извлечения значений для их вывода. После этого set-функции пытаются присвоить элементам **hour** и **second** недопустимые значения, а элементу **minute** допустимое значение, после чего get-функции возвращают значения для вывода. Выходные данные программы подтверждают, что недопустимые значения приводят к присваиванию элементам данных нулевых значений. И наконец, программа устанавливает время в **11:58:00** и увеличивает на 3 значение **minute** путем вызова функции **incrementMinutes**. Функция **incrementMinutes** не является элементом класса и использует для увеличения элементов данных элементы-функции get... и set.... Этот метод работает, но приводит к снижению производительности, связанному с неоднократными вызовами функций. В следующей главе мы обсудим понятие дружественных функций в качестве одного из способов устранения подобных издержек.

Наличие в классе set-функций представляет программисту определенную гибкость в написании конструкторов.

```
// TIME3.H
// Объявление класса Time.
// Элементы-функции определены в TIME3.CPP

// не допускает повторных включений заголовочного файла
#ifndef TIME3_H
#define TIME3_H

class Time {
public:
   Time(int = 0, int = 0, int = 0);  // конструктор

   // set-функции
   void setTime(int, int, int); // устанавливает hour, minute,
                                // second
   void setHour(int);     // устанавливает hour
   void setMinute(int);   // устанавливает minute
   void setSecond(int);   // устанавливает second

   // get-функции
   int getHour();     // возвращает hour
   int getMinute();   // возвращает minute
   int getSecond();   // возвращает second
```

Рис. 16.10. Объявление класса **Time** (часть 1 из 6)

```
   void printMilitary();  // выводит время в военном формате
   void printStandard();  // выводит время в стандартном формате
private:
   int hour;     // 0 - 23
   int minute;   // 0 - 59
   int second;   // 0 - 59
};

#endif
```

Рис. 16.10. Объявление класса **Time** (часть 2 из 6)

```
// TIME3.CPP
// Определения элементов-функций класса Time

#include "time3.h"
#include <iostream.h>

// Конструктор для инициализации закрытых данных.
// Значения по умолчанию равны 0 (см. определение класса).
Time::Time(int hr, int min, int sec)
{
   hour = ( hr >= 0 && hr < 24) ? hr : 0;
   minute = ( min >= 0 && min < 60) ? min : 0;
   second = ( sec >= 0 && sec < 60) ? sec : 0;
}

// Устанавливает значения hour, minute и second.
void Time::setTime(int h, int m, int s)
{
   hour = ( h >= 0 && h < 24) ? h : 0;
   minute = ( m >= 0 && m < 60) ? m : 0;
   second = ( s >= 0 && s < 60) ? s : 0;
}
// Устанавливает значение hour
void Time::setHour(int h)
   { hour = ( h >= 0 && h < 24) ? h : 0; }

// Устанавливает значение minute
void Time::setMinute(int m)
   { minute = ( m >= 0 && m < 60) ? m : 0; }

// Устанавливает значение second
void Time::setSecond(int s)
   { second = ( s >= 0 && s < 60) ? s : 0; }

// Получает значение hour
int Time::detHour()   { return hour; }

// Получает значение minute
int Time::getMinute()   { return minute; }

// Получает значение second
int Time::getSecond()   { return second; }
```

Рис. 16.10. Определения элементов-функций класса **Time** (часть 3 из 6)

```
// Выводит время в военном формате: HH:MM:SS
void Time::printMilitary()
{
   cout << ( hour < 10 ? "0" : "") << hour << ":"
        << ( minute < 10 ? "0" : "") << minute << ":"
        << ( second < 10 ? "0" : "") << second;
}
// Выводит время в стандартном формате: HH:MM:SS AM (или PM)
void Time::printStandard()
{
   cout << (( hour == 0 || hour == 12) ? 12 : hour % 12) << ":"
        << ( minute < 10 ? "0" : "") << minute << ":"
        << ( second < 10 ? "0" : "") << second
        << ( hour < 12 ? " AM" : " PM");
}
```

Рис. 16.10. Определения элементов-функций класса **Time** (часть 4 из 6)

```
// FIG16_10.CPP
// Демонстрация set- и get-функций класса Time
#include <iostream.h>
#include "time3.h"

void incrementMinutes(Time &, const int);

main()
{
   Time t;

   t.setHour(17);
   t.setMinute(34);
   t.setSecond(25);

   cout << "Result of setting all valid values:\n Hour: "
        << t.getHour()
        << " Minute: " << t.getMinute()
        << " Second: " << t.getSecond() << "\n\n";

   t.setHour(234);  // недопустимое значение hour установлено в 0
   t.setMinute(43);
   t.setSecond(6373); // недопустимое значение second
                      // установлено в 0

   cout << "Result of attempting to set invalid hour and"
        << " second: \n Hour: " << t.getHour()
        << " Minute: " << t.getMinute()
        << " Second: " << t.getSecond() << "\n\n";

   t.setTime(11, 58, 0);
   incrementMinutes(t, 3);
   return 0;
}

void incrementMinutes(Time &tt, const int count)
{
```

Рис. 16.10. Использование set- и get-функций (часть 5 из 6)

```
   cout << "Incrementing minute " << cout
        << " times:\nStart time: ";
   tt.printStandard();

   for (int i = 1; i <= count; i++) {
      tt.setMinute(( tt.getMinute() + 1) % 60);

      if (tt.getMinute() == 0)
         tt.setHour ((tt.getHour() + 1) % 24);

      cout << "\nminute + 1: ";
      tt.printStandard();
   }
cout << endl;
}
```

```
Result of setting all valid values:
  Hour: 17   Minute: 34   Second: 25

Result of attempting to set invalid hour and second:
  Hour: 0   Minute: 43   Second: 0

Incrementing minute 3 times:
Start time: 11:58:00 AM
minute + 1: 11:59:00 AM
minute + 1: 12:00:00 PM
minute + 1: 12:01:00 PM
```

Рис. 16.10. Использование set- и get-функций (часть 6 из 6)

Распространенная ошибка программирования 16.8.

Конструктор может вызывать другие элементы-функции класса, например, set- или get-функции, однако, поскольку конструктор инициализирует объект, элементы данных могут еще не находиться в целостном состоянии. Использование элементов данных до их надлежащей инициализации может приводить к ошибкам.

Set-функции, безусловно, важны с точки зрения конструирования программного обеспечения, поскольку они могут выполнять проверку корректности данных. Set- и get-функции имеют и другие достоинства, важные в *отношении конструирования программного обеспечения*.

Общее методическое замечание 16.16

Доступ к закрытым данным посредством функций-элементов set... и get... не только защищает элементы данных от присваивания им недопустимых значений, но также изолирует клиентов класса от представления данных. Таким образом, если представление данных по некоторым причинам меняется (часто в целях уменьшения количества требуемой памяти или повышения производительности), требуется изменить только элементы-функции — клиенты класса не нуждаются в изменениях до тех пор, пока интерфейс, предоставляемый этими элементами-функциями, остается тем же самым. Однако может потребоваться перекомпиляция программ-клиентов.

16.15. Скрытая ловушка: возвращение ссылки на закрытый элемент данных

Ссылка на объект является псевдонимом для самого объекта и, следовательно, она может стоять в левой части оператора присваивания. Другими словами, ссылка вполне приемлема в качестве *lvalue*, способного принимать значение. Один из способов воспользоваться этим ее свойством (к сожалению!) — это заставить открытую функцию-элемент класса возвращать ссылку на закрытый элемент данных этого класса.

На рис. 16.11 используется упрощенная версия класса **Time** для демонстрации возврата ссылки на закрытый элемент данных. Такого рода возвращаемое значение фактически делает из вызова функции **badSetHour** псевдоним для закрытого элемента данных **hour**! Этот вызов функции можно использовать любым из способов, возможных для открытого элемента данных, в том числе в качестве *lvalue* в операторе присваивания!

```
// TIME4.H
// Объявление класса Time.
// Элементы-функции определены в TIME4.CPP

// не допускает повторных включений заголовочного файла
#ifndef TIME4_H
#define TIME4_H

class Time {
public:
   Time(int = 0, int = 0, int = 0);
   void setTime(int, int, int);
   int getHour();
   int &badSetHour(int);  // ОПАСНОЕ возвращение ссылки
private:
   int hour;
   int minute;
   int second;
};

#endif

// TIME4.CPP
// Определения элементов-функций класса Time.
#include "time4.h"
#include <iostream.h>

// Конструктор для инициализации закрытых данных.
// Вызывает элемент-функцию setTime для установки переменных.
// Значения по умолчанию равны 0 (см. определение класса).
Time::Time(int hr, int min, int sec)
   { setTime ( hr, min, sec); }

// Устанавливает значения hour, minute и second.
void Time::setTime ( int h, int m, int s)
{
   hour = (h >= 0 && h < 24) ? h : 0;
```

Рис. 16.11. Возвращение ссылки на закрытый элемент данных (часть 1 из 2)

```
   minute = ( m >= 0 && m < 60) ? m : 0;
   second = ( s >= 0 && s < 60) ? s : 0;
}

// Получает значение hour
int Time::getHour()  { return hour; }

// ПЛОХОЙ СТИЛЬ ПРОГРАММИРОВАНИЯ:
// Возвращение ссылки на закрытый элемент данных.
int &Time::badSetHour(int hh)
{
   hour = ( hh >= 0 && hh < 24) ? hh : 0;

   return hour;  // ОПАСНОЕ возвращение ссылки
}
// FIG16_11.CPP
// Демонстрация открытой функции-элемента, которая
// возвращает ссылку на закрытый элемент данных.
// Для этого примера класс Time был несколько урезан.

#include <iostream.h>
#include "time4.h"

main()
{
   Time t;
   int &hourRef = t.badSetHour(20);

   cout << "Hour before modification: "
        << hourRef << '\n' ;
   hourRef = 30; // модификация недопустимым значением
   cout << "Hour after modification: "
        << t.getHour() << '\n';

   // Опасность: Вызов функции, возвращающей
   // ссылку, может быть использован в качестве lvalue.
   t.badSetHour(12) = 74;
   cout << "\n*********************************\n"
        << "BAD PROGRAMMING PRACTICE!!!!!!!!!\n"
        << "badSetHour as an lvalue, Hour: "
        << t.getHour()
        << "\n*********************************\n";

   return 0;
}
```

```
Hour before modification: 20
Hour after modification: 30

*********************************
BAD PROGRAMMING PRACTICE!!!!!!!!!
badSetHour as an lvalue, hour: 74
*********************************
```

Рис. 16.11. Возвращение ссылки на закрытый элемент данных (часть 2 из 2)

Хороший стиль программирования 16.12

Никогда не допускайте, чтобы открытая функция-элемент возвращала неконстантную ссылку (или указатель) на закрытый элемент данных. Возвращение такого рода ссылки нарушает инкапсуляцию данных класса.

Выполнение программы начинается с объявления объекта **t** класса **Time** и ссылки **hourRef**, которая инициализируется ссылкой, возвращаемой при вызове **t.badSetHour(20)**. Программа отображает на экране значение псевдонима **hourRef**. Затем этот псевдоним используется для присваивания элементу **hour** значения 30 (недопустимое значение) и его значение снова отображается на экране. И наконец, сам вызов функции используется в качестве *lvalue* и ему присваивается значение 74 (еще одно недопустимое значение) и это значение отображается на экране.

16.16. Присваивание по умолчанию путем поэлементного копирования

Операция присваивания (=) используется для присваивания некоторого объекта другому объекту того же типа. Такое присваивание обычно выполняется при помощи *поэлементного копирования* — каждый элемент одного объекта по отдельности копируется в такой же элемент в другом объекте (см. рис. 16.12). (Замечание: поэлементное копирование может вызывать серьезные проблемы, если оно используется с классом, элементы данных которого содержат динамически выделяемую память; в главе 18, «Перегрузка операций», мы будем обсуждать эти проблемы и покажем способы их решения.)

Объекты можно передавать функциям в качестве аргументов и можно возвращать их из функций. По умолчанию такая передача параметров (и их возвращение) происходит по значению — передается или возвращается копия объекта (мы представим несколько примеров в главе 18, «Перегрузка операций»).

Совет по повышению эффективности 16.4

Передача объектов по значению хороша с точки зрения безопасности, поскольку вызываемая функция не имеет доступа к исходному объекту, однако вызов по значению может снижать производительность системы в случае создания копии большого объекта. Объект может передаваться по ссылке путем передачи либо указателя на объект, либо ссылки на него. Передача по ссылке повышает производительность, но слабее с точки зрения безопасности, поскольку вызываемой функции предоставляется доступ к исходному объекту. Безопасной альтернативой является передача объекта по ссылке на константу.

16.17. Повторное использование программного обеспечения

Программисты, пишущие объектно-ориентированные программы, концентрируют внимание на реализации полезных классов. Существует многообещающая возможность создания каталогов классов, к которым могут иметь доступ большие группы программистов. Уже имеется множество *библиотек классов*, и по всему миру разрабатываются новые библиотеки. Прилагаются все усилия к

тому, чтобы сделать эти библиотеки широко доступными. В этом случае программное обеспечение могло бы создаваться на основе уже существующих точно определенных, тщательно проверенных, хорошо документированных, переносимых и широкодоступных компонентов. Такого рода *повторное использование программного кода* может ускорить разработку мощного и высококачественного программного обеспечения. Становится возможной *ускоренная разработка прикладных программ* (*RAD* — Rapid Application Development).

```
// FIG16_12.CPP
// Демонстрация возможности присваивания по умолчанию
// объектов класса друг другу с помощью поэлементного копирования
#include <iostream.h>

// Простой класс Date
class Date {
public:
   Date(int = 1, int = 1, int = 1990); // конструктор
                                       // по умолчанию
   void print();
private:
   int month;
   int day;
   int year;
};

// Простой конструктор класса Date без проверки диапазона
Date::Date(int m, int d, int y)
{
   month = m;
   day = d;
   year = y;
}

// Выводит объект Date в формате mm-dd-yyyy
void Date::print()
   { cout << month << '-' << day << '-' << year; }

main()
{
   Date d1 ( 7, 4, 1993), d2;  // d2 принимает значение
                               // по умолчанию 1/1/90

   cout << "d1 = ";
   d1.print();
   cout << "\nd2 = " ;
   d2.print();

   d2 = d1;  // присваивание с помощью поэлементного копирования
             // по умолчанию
   cout << "\n\nAfter default memberwise copy, d2 = ";
   d2.print();
   cout << endl;

   return 0;
}
```

Рис. 16.12. Присваивание одного объекта другому при помощи поэлементного копирования по умолчанию (часть 1 из 2)

```
d1 = 7-4-1993
d2 = 1-1-1990

After default memberwise copy,  d2 = 7-4-1993
```

Рис. 16.12. Присваивание одного объекта другому при помощи поэлементного копирования по умолчанию (часть 2 из 2)

Однако, прежде чем потенциал повторного использования программного обеспечения можно будет реализовать полностью, должны быть решены существенные проблемы. Нам потребуются схемы каталогизации, схемы патентования, механизмы защиты, гарантирующие достоверность исходных копий классов, схемы описания, позволяющие проектировщикам новых систем определять, соответствуют ли их потребностям существующие объекты, механизмы просмотра, позволяющие определять, какие классы доступны и насколько близко они соответствуют требованиям разработчика программного обеспечения и т.д. Предстоит решить множество интересных научно-исследовательских задач. И эти задачи будут решены, поскольку потенциальное значение их решения огромно.

Резюме

- Структура представляет собой агрегатный тип данных, создаваемый на основе объектов других типов.
- Ключевое слово **struct** начинает определение структуры. Тело структуры ограничивается фигурными скобками ({ и }). Определение структуры должно завершаться точкой с запятой.
- Имя-этикетка структуры может быть использовано для объявления переменных структурного типа.
- Определения структур не резервируют места в памяти; они создают новые типы данных, которые используются для объявления переменных.
- Для доступа к элементам структуры (или класса) используются операции доступа к элементам — операция-точка (.) и операция-стрелка (->). Операция-точка обращается к элементу структуры посредством имени переменной-объекта или ссылки на объект. Операция-стрелка обращается к элементу структуры посредством указателя на объект.
- Отрицательными сторонами создания новых типов данных с помощью ключевого слова **struct** языка C являются возможность получения неинициализированных данных; возможность некорректной инициализации; все программы, использующие объекты типа **struct** в стиле языка C, должны быть изменены при изменении реализации этого типа **struct**; также не предусматривается защитных мер, позволяющих гарантировать, что данные хранятся в целостном состоянии и содержат правильные значения.
- Классы дают возможность программисту моделировать объекты, обладающие свойствами и поведением. Классы в языке C++ могут определяться с помощью ключевых слов **class** и **struct**, однако обычно для этой цели используется ключевое слово **class**.

- Имя класса может быть использовано для объявления объектов этого класса.
- *Определение класса начинается с ключевого слова* **class**. *Тело определения* класса ограничивается фигурными скобками ({ и }). Определение класса завершается точкой с запятой.
- Все элементы данных и элементы-функции, объявленные в классе после спецификатора доступа **public**:, являются видимыми для всех функций, имеющих доступ к объекту этого класса.
- Все элементы данных и элементы-функции, объявленные в классе после спецификатора доступа **private**:, являются видимыми только для друзей класса и других его элементов.
- Спецификаторы доступа к элементам всегда оканчиваются двоеточием (:) и могут многократно появляться в определении класса.
- Закрытые данные недоступны за пределами класса.
- Говорят, что реализация класса скрыта от его клиентов, или «инкапсулирована».
- Конструктор — это специальная функция-элемент с тем же именем, что и класс, которая используется для инициализации элементов объекта класса. Конструкторы вызывается при создании объекта их класса.
- Функция с тем же именем, что и класс, которому предшествует символ тильды (~), называется деструктором.
- Множество открытых элементов-функций класса называется интерфейсом класса, или открытым интерфейсом класса.
- При определении элемента-функции за пределами области определения класса ее имени предшествует имя класса и двухместная операция разрешения области действия (::).
- Элементы-функции, определяемые за пределами области определения класса с использованием операции разрешения области действия, находятся внутри области действия этого класса.
- Элементы-функции, определяемые в определении класса, автоматически становятся встроенными. Компилятор оставляет за собой право не делать встроенной любую функцию.
- Вызов элементов-функций является более кратким, чем вызов функций в процедурных языках программирования, поскольку для элементов-функций предполагается, что необъявленные имена являются элементами класса, в котором определена эта функция-элемент.
- Внутри области действия класса на элементы класса можно ссылаться просто по их именам. Вне области действия класса ссылка на элементы класса происходит либо посредством имени объекта, либо посредством ссылки на объект, либо посредством указателя на объект.
- Для доступа к элементам класса используются операции выбора элемента . и ->.

- Фундаментальным принципом систематического конструирования программного обеспечения является отделение интерфейса от реализации с целью упрощения модификации программ.
- Обычно определения классов помещают в заголовочные файлы, а определения его элементов-функций помещают в файлы исходного кода с тем же базовым именем.
- Режимом доступа по умолчанию для классов является **private:**, поэтому все элементы после заголовка класса и до первой метки доступа считаются закрытыми.
- Открытые элементы класса дают представление об услугах, предоставляемых классом.
- Доступ к закрытым элементам класса может тщательно контролироваться путем использования элементов-функций, называемых функциями доступа. Если класс хочет дать своим клиентам возможность читать закрытые данные, он может определять соответствующую get-функцию. Чтобы дать клиентам возможность изменять закрытые данные, класс может иметь соответствующую set-функцию.
- Обычно элементы данных класса делают закрытыми, а его элементы-функции — открытыми. Некоторые элементы-функции остаются закрытыми и служат в качестве сервисных для других функций класса.
- Элементы данных класса не могут быть инициализированы в его определении. Они должны инициализироваться в конструкторе, либо им могут быть присвоены значения позже после создания соответствующих объектов.
- Конструкторы могут быть перегруженными.
- Как только объект класса корректно инициализирован, все элементы-функции, манипулирующие этим объектом, должны гарантировать, что объект остается в целостном состоянии.
- При объявлении объекта класса могут быть заданы инициализаторы. Эти инициализаторы передаются в конструктор класса.
- В конструкторах могут быть специфицированы аргументы по умолчанию.
- В конструкторах не может быть специфицирован тип возвращаемого значения, а также не могут предприниматься попытки возвратить значение.
- Если для класса не определено ни одного конструктора, компилятор создает конструктор по умолчанию. Предусмотренный компилятором конструктор по умолчанию не выполняет никакой инициализации, поэтому после создания объекта не гарантируется целостность его состояния.
- Деструктор класса вызывается автоматически, когда объект класса выходит из области действия. Сам деструктор фактически не разрушает объекта, однако он выполняет заключительную «приборку», прежде чем системе будет возвращена выделенная объекту память.
- Деструкторы не принимают аргументов и не возвращают значений. У класса может быть только один деструктор.

- Операция присваивания (=) используется для присваивания некоторого объекта другому объекту того же типа. Такое присваивание обычно выполняется посредством поэлементного копирования — каждый элемент одного объекта по отдельности копируется в такой же элемент в другом объекте.

Терминология

class
get-функция
inline (встроенная) функция-элемент
private:
protected:
public:
set-функция
абстрактный тип данных (ADT)
абстракция данных
автоматический локальный объект
атрибут
вхождение в область действия
выход из области действия
глобальный объект
деструктор
динамические объекты
заголовочный файл
имя-этикетка класса
инициализатор элемента
инициализация объекта класса
инкапсуляция
интерфейс класса
клиент класса
конструктор
конструктор по умолчанию
метод
область действия класса
область действия файла
объект
Объектно-ориентированное программирование (OOP)
объявление класса
операция взятия адреса &
операция выбора элемента класса (.)
операция присваивания (=)
операция разрешения области действия ::
операция ссылки &
определение класса
открытый интерфейс класса
поведение
повторно используемый код
повторное использование программного кода
поэлементное копирование
предикатная функция
представитель класса
принцим минимума привилегий
процедурное программирование
расширяемость
реализация класса
сервисная функция
создание экземпляра объекта
сокрытие данных
сокрытие информации
сообщение
спецификаторы доступа к элементам
статический локальный объект
тильда (~) в имени деструктора
тип данных
тип, определяемый пользователем
управление доступом к элементам
ускоренная разработка прикладных программ (RAD)
услуги, предоставляемые классом
файл исходного кода
функция доступа
функция-элемент
целостное состояние элементов данных
элемент данных

Распространенные ошибки программирования

16.1. Забывают о точке с запятой в конце определения класса.

16.2. Попытка явной инициализации элемента данных класса.

16.3. Попытка перегрузить элемент-функцию класса функцией, не находящейся в области действия этого класса.

16.4. Попытка функции, не являющейся элементом-функцией данного класса (или его другом), получить доступ к закрытому элементу этого класса.

16.5. Попытка явной инициализации элемента данных класса в определении класса.

16.6. Попытка объявить для конструктора тип возвращаемого значения и/или попытка возвратить значение из конструктора.

16.7. Попытка передать параметры в деструктор, возвратить значение из деструктора или перегрузить деструктор.

16.8. Конструктор может вызывать другие элементы-функции класса, например, set- или get-функции, однако, поскольку конструктор инициализирует объект, элементы данных могут еще не находиться в целостном состоянии. Использование элементов данных до их надлежащей инициализации может приводить к ошибкам.

Хороший стиль программирования

16.1. Используйте в определении класса каждый спецификатор доступа только один раз из соображений ясности и удобочитаемости программы. Помещайте открытые элементы в начале определения, чтобы их было проще найти.

16.2. Определяйте все элементы-функции, кроме самых маленьких, вне определения класса. Это способствует отделению интерфейса класса от его реализации.

16.3. Используйте директивы препроцессора **#ifndef**, **#define** и **#endif**, чтобы не допустить повторного включения в программу заголовочных файлов.

16.4. Используйте имя заголовочного файла с подчеркиванием вместо точки в директивах препроцессора **#ifndef** и **#define** заголовочных файлов.

16.5. Если вы хотите сначала перечислить закрытые элементы в объявлении класса, явно используйте метку **private:**, несмотря на то, что тип доступа **private:** принят по умолчанию. Это делает программу более понятной. Мы предпочитаем сначала перечислять элементы класса с типом доступа **public:**.

16.6. Несмотря на то, что метки **public:** и **private:** могут повторяться и чередоваться, перечислите сначала в одной группе все открытые элементы класса, а затем перечислите в другой группе все его закрытые элементы. Это акцентирует внимание пользователя на открытом интерфейсе класса, а не на его реализации.

16.7. Использование каждого из спецификаторов доступа к элементам **public:**, **protected:** и **private:** только один раз в определении класса помогает избежать путаницы.

16.8. Когда это целесообразно (почти всегда), определяйте конструктор, чтобы гарантировать надлежащую инициализацию каждого объекта осмысленными значениями.

16.9. Любая функция-элемент (или дружественная функция), изменяющая закрытые элементы данных объекта, должна гарантировать, что данные остаются в целостном (действительном) состоянии.

16.10. Всегда предоставляйте конструктор, который выполняет соответствующую инициализацию объектов своего класса.

16.11. Элементы-функции, присваивающие значения закрытым данным, должны проверять корректность вновь вводимых значений; в случае их некорректности set-функции должны приводить закрытые элементы в некоторое состояние, гарантирующее целостность данных.

16.12. Никогда не допускайте, чтобы открытая функция-элемент возвращала неконстантную ссылку (или указатель) на закрытый элемент данных. Возвращение такого рода ссылки нарушает инкапсуляцию данных класса.

Советы по повышению эффективности

16.1. Обычно структуры передаются по значению. Чтобы избежать накладных расходов на их копирование, передавайте структуры по ссылке.

16.2. Чтобы избежать накладных расходов при передаче параметров по значению и вместе с тем извлечь выгоду из защиты от изменения исходных данных, передавайте параметры большого размера в виде константных ссылок.

16.3. Определение элемента-функции небольшого размера внутри определения класса автоматически делает эту функцию встроенной (если с этим согласен компилятор). Это может повысить производительность системы, однако не всегда согласуется с принципами систематической разработки программного обеспечения.

16.4. Передача объектов по значению хороша с точки зрения безопасности, поскольку вызываемая функция не имеет доступа к исходному объекту, однако вызов по значению может снижать производительность системы в случае создания копии большого объекта. Объект может передаваться по ссылке путем передачи либо указателя на объект, либо ссылки на него. Передача по ссылке повышает производительность, но слабее с точки зрения безопасности, поскольку вызываемой функции предоставляется доступ к исходному объекту. Безопасной альтернативой является передача объекта по ссылке на константу.

Общие методические замечания

16.1. Важно писать понятные и простые для сопровождения программы. Изменения в программах являются скорее правилом, чем исключением. Программисты должны предвидеть заранее, что их код будет модифицироваться. Как мы увидим, классы способствуют модифицируемости программ.

16.2. Клиенты класса использует этот класс, не зная внутренних деталей его реализации. При изменении реализации класса (например, из

соображений эффективности) клиентов класса изменять не требуется. Это значительно облегчает модификацию программных систем.

16.3. Обычно элементы-функции состоят из всего лишь из нескольких строк кода, поскольку не требуется никакой логики для проверки корректности значений элементов данных.

16.4. Клиенты класса имеет доступ к интерфейсу класса, но они не должны иметь доступа к его реализации.

16.5. Объявление элементов-функций внутри определения класса с последующим определением их вне определения этого класса отделяет интерфейс класса от его реализации. Это способствует систематизации разработки программного обеспечения.

16.6. Объектно-ориентированный подход к программированию часто приводит к упрощению вызовов функций из-за уменьшения числа передаваемых параметров. Это преимущество объектно-ориентированного программирования является следствием того, что инкапсуляция элементов данных и элементов-функций внутри объекта дает элементам-функциям право обращаться к этим элементам данных.

16.7. Помещайте объявление класса в заголовочный файл, который должен быть включен любым клиентом, желающим использовать этот класс. Это — открытый интерфейс класса. Помещайте определения элементов-функций этого класса в исходный файл. Это — реализация класса.

16.8. Клиентам класса не обязательно видеть его исходный код для того, чтобы пользоваться этим классом. Однако, клиентам необходимо иметь возможность компоноваться с объектным кодом класса.

16.9. Информация, являющаяся важной для интерфейса класса, должна включаться в заголовочный файл. Информация, которая будет использоваться только внутри класса и не потребуется клиентам класса, должна находиться в неопубликованном исходном файле. Это еще один пример принципа минимума привилегий.

16.10. C++ поощряет независимость программ от реализации. Когда реализация класса, который используется в независимом от реализации коде, изменяется, этот код не нуждается в модификации, однако может потребоваться его перекомпиляция.

16.11. Оставляйте все элементы данных класса закрытыми. Предусмотрите публичные элементы-функции для установки значений закрытых элементов данных и получения значений закрытых элементов данных. Такая архитектура помогает скрывать реализацию класса от его клиентов, что уменьшает количество ошибок и способствует модифицируемости программы.

16.12. Проектировщики классов используют закрытые, защищенные и открытые элементы для реализации принципов сокрытия информации и минимальных привилегий.

16.13. Задание закрытого доступа для элементов данных класса и открытого доступа для его элементов-функций облегчает отладку, поско-

льку проблемы, связанные с манипулированием данными, ограничиваются либо элементами-функциями класса, либо его друзьями.

16.14. Элементы-функции имеют тенденцию подразделяться на ряд различных категорий: функции, которые считывают и возвращают значения закрытых элементов данных, функции, которые устанавливают значения закрытых элементов данных, функции, которых реализуют характерные особенности класса, и функции, которые выполняют разнообразную механическую работу для класса, такую, как инициализация объектов класса, присваивание объектов, преобразование между классами и встроенными типами данных или между классами и другими классами, и управление памятью для объектов класса.

16.15. Закрытие элементов данных и контроль доступа (особенно доступа для записи) к этим элементам посредством открытых элементов-функций помогает гарантировать целостность данных.

16.16. Доступ к закрытым данным посредством функций-элементов set... и get... не только защищает элементы данных от присваивания им недопустимых значений, но также изолирует клиентов класса от представления данных. Таким образом, если представление данных по некоторым причинам меняется (часто в целях уменьшения количества требуемой памяти или повышения производительности), требуется изменить только элементы-функции — клиенты класса не нуждаются в изменениях до тех пор, пока интерфейс, предоставляемый этими элементами-функциями, остается тем же самым. Однако может потребоваться перекомпиляция программ-клиентов.

Упражнения для самоконтроля

16.1. Заполните пропуски в каждом из следующих предложений:

a) Ключевое слово _____ предваряет определение структуры.

b) Доступ к элементам класса осуществляется посредством операции _____ совместно с объектом класса или посредством операции _____ совместно с указателем на объект класса.

c) Элементы класса, определенные как _____, доступны только для элементов-функций класса и друзей класса.

d) _____ является специальной функцией-элементом, которая используется для инициализации элементов данных класса.

e) Доступом по умолчанию для элемента класса является _____.

f) _____ функция используется для присваивания значений закрытым элементам данных класса.

g) Для присваивания объекта класса другому объекту того же класса может быть использовано _____.

h) Элементы-функции класса обычно делают _____, а его элементы данных — _____.

i) _____ функция используется для извлечения значений закрытых данных класса.

j) Множество открытых элементов-функций класса называется _____ класса.

k) Говорят, что реализация класса скрыта от его клиентов, или _____.

l) Для определения класса могут быть использованы ключевые слова _____ и _____.

m) Элементы класса, специфицированные как _____, доступны везде, где объект класса находится в области действия.

16.2. Найдите ошибку(и) в каждом из следующих случаев и объясните, как ее исправить.

a) Предположим, что в классе **Time** объявлен следующий прототип:

```
void ~Time(int);
```

b) Далее следует фрагмент определения класса **Time**.

```
class Time {
public:
   // прототипы функций
private:
   int hour = 0;
   int minute = 0;
   int second = 0;
};
```

c) Предположим, что в классе **Employee** объявлен следующий прототип:

```
int Employee(const char *, const char *);
```

Ответы к упражнениям для самоконтроля

16.1. a) **struct**. b) -точки (.), -стрелки (->). c) **private**. d) Конструктор. e) **private**. f) set-. g) поэлементное копирование по умолчанию. h) **private**, **public**. i) get-. j) интерфейсом. k) инкапсулирована. l) **public**. m) **class**, **struct**.

16.2. a) Ошибка: деструкторам не разрешается возвращать значение или принимать параметры.

Исправление: удалите из объявления тип возвращаемого значения **void** и параметр **int**.

b) Ошибка: элементы данных не могут явно инициализироваться при определении класса.

Исправление: удалите из определения класса явную инициализацию и инициализируйте элементы данных в конструкторе.

c) Ошибка: возвращение конструкторами значений недопустимо.

Исправление: удалите из объявления тип возвращаемого значения **int**.

Упражнения

16.3. Для чего нужна операция разрешения области действия?

16.4. Сравните понятия структуры и класса в языке C++.

16.5. Напишите конструктор, который может использовать текущее время, возвращаемое функцией **time()** — объявленной в заголовочном файле **time.h** стандартной библиотеки языка C — для инициализации объекта класса **Time**.

16.6. Создайте класс с именем **Complex** для выполнения арифметических операций с комплексными числами. Напишите программу для тестирования вашего класса.

Комплексные числа имеют вид

```
realPart + imaginaryPart * i,
```

где i равно $\sqrt{-1}$.

Используйте переменные с плавающей точкой для представления закрытых данных класса. Определите конструктор, который дает возможность инициализировать объект класса при его объявлении. Конструктор должен содержать значения по умолчанию на случай отсутствия инициализаторов. Предусмотрите элементы-функции для каждого из следующих действий:

a) Сложение двух чисел типа **Complex**: складываются вместе их вещественные и мнимые части.

b) Вычитание двух чисел типа **Complex**: вещественная часть правого операнда вычитается из вещественной части левого операнда и мнимая часть правого операнда вычитается из мнимой части левого.

c) Вывод чисел типа **Complex** в виде **(a, b)**, где **a** — его вещественная часть и **b** — мнимая часть.

16.7. Создайте класс с именем **RationalNumber** для выполнения арифметических действий с дробями. Напишите программу для тестирования вашего класса.

Используйте целочисленные переменные для представления закрытых данных класса — числителя и знаменателя. Предусмотрите конструктор, который дает возможность инициализировать объект класса при его объявлении. Конструктор должен содержать значения по умолчанию на случай отсутствия инициализаторов и должен хранить дробь в несокращаемом виде (т.е. для дроби 2/4 в объекте хранится 1 в числителе и 2 в знаменателе). Предусмотрите открытые элементы-функции для каждого из следующих действий:

a) Сложение двух рациональных чисел. Результат должен быть сохранен в несокращаемом виде.

b) Вычитание двух рациональных чисел. Результат должен быть сохранен в несокращаемом виде.

c) Умножение двух рациональных чисел. Результат должен быть сохранен в несокращаемом виде.

d) Деление двух рациональных чисел. Результат должен быть сохранен в несокращаемом виде.

e) Вывод рациональных чисел в виде **a/b**, где **a** является числителем и **b** — знаменателем.

f) Вывод рациональных чисел в формате с плавающей точкой.

16.8. Измените класс **Time** на рис. 16.10 так, чтобы он включал элемент-функцию **tick**, которая увеличивает на одну секунду время, хранимое в объекте **Time**. Объект **Time** всегда должен оставаться в корректном состоянии. Напишите программу, тестирующую элемент-функцию **tick** в цикле, который выводит время в стандартном формате на каждой итерации цикла для иллюстрации правильной работы элемента-функции **tick**. Убедитесь, что проверены следующие случаи:

a) Переход при увеличении времени к следующей минуте.

b) Переход при увеличении времени к следующему часу.

c) Переход при увеличении времени к следующему дню (т.е. от 11:59:59 PM к 12:00:00 AM).

16.9. Измените класс **Date** на рис. 16.12 так, чтобы в нем выполнялась проверка на наличие ошибок значений инициализаторов для элементов данных **month**, **day** и **year**. Кроме того, напишите элемент-функцию **nextDay** для увеличения даты на единицу. Объект **Date** должен всегда оставаться в корректном состоянии. Напишите программу, тестирующую функцию **nextDay** в цикле, который выводит дату на каждой итерации цикла для иллюстрации правильной работы функции **nextDay**. Убедитесь, что проверены следующие случаи:

a) Переход при увеличении даты к следующему месяцу.

b) Переход при увеличении даты на следующему году.

16.10. Объедините модифицированные классы **Time** из упражнения 16.8 и **Date** из упражнения 16.9 в один класс с именем **DateAndTime** (в главе 19 мы обсудим наследование, которое даст нам возможность быстро решить эту задачу без изменения существующих определений классов). Измените функцию **tick** таким образом, чтобы она вызывала функцию **nextDay**, если при увеличении времени мы переходим к следующему дню. Измените функции **printStandard** и **printMilitary** так, чтобы в дополнение ко времени они выводили дату. Напишите программу для тестирования нового класса **DateAndTime**. Специально проверьте переход к следующему дню при увеличении времени.

глава

17

Классы: часть II



Цели

- Научиться динамически создавать и уничтожать объекты.
- Научиться определять константные объекты и константные функции-элементы.
- Понять смысл определения дружественных функций и классов.
- Понять принципы использования статических элементов данных и функций-элементов.
- Познакомиться с различными типами контейнерных классов.
- Научиться разрабатывать классы итераторов, перебирающих элементы контейнерного класса.
- Изучить применение указателя this.
- Научиться создавать и использовать шаблоны классов.

Содержание

17.1. Введение
17.2. Константные объекты и константные элементы-функции
17.3. Композиция: классы в качестве элементов других классов
17.4. Дружественные функции и дружественные классы
17.5. Указатель this
17.6. Динамическое распределение памяти с помощью операций new и delete
17.7. Статические элементы класса
17.8. Абстракция данных и сокрытие информации
17.8.1. Пример: абстрактный тип данных — массив
17.8.2. Пример: абстрактный тип данных — строка
17.8.3. Пример: абстрактный тип данных — очередь
17.9. Контейнерные классы и итераторы
17.10. Шаблоны классов

Резюме • Распространенные ошибки программирования • Хороший стиль программирования • Советы по повышению эффективности • Советы по переносимости программ • Общие методические замечания • Упражнения для самоконтроля • Ответы к упражнениям для самоконтроля • Упражнения

17.1. Введение

В этой главе мы продолжим изучение классов и абстракции данных. Мы обсудим несколько более сложную тематику и заложим фундамент для обсуждения классов и перегрузки операций в главе 18. Обсуждение, проводимое в главах с 16 по 18, побуждает программистов к использованию объектов, показывая ряд их достоинств. Затем в главах 19 и 20 вводятся понятия наследования и полиморфизма, составляющих подлинную суть объектно-ориентированного программирования.

17.2. Константные объекты и константные элементы-функции

Мы неоднократно акцентировали внимание на *принципе минимума привилегий* как на одном из наиболее фундаментальных принципов систематического конструирования программного обеспечения. Давайте рассмотрим один из путей применения этого принципа к объектам.

Некоторые объекты должны быть модифицируемыми, а другие нет. Программист может использовать ключевое слово **const**, чтобы указать, что объект не является модифицируемым и что любая попытка изменения этого объекта является ошибкой. Например,

```
const Time noon(12, 0, 0);
```

объявляет константный объект **noon** класса **Time** и инициализирует его 12 часами дня.

Общее методическое замечание 17.1

Объявление объекта с модификатором **const** способствует последовательному проведению в жизнь принципа минимума привилегий. Случайные попытки изменить объект выявляются во времена компиляции и не приводят к ошибкам времени выполнения.

Компиляторы C++ соблюдают объявления **const** настолько строго, что ими полностью отвергаются любые вызовы элементов-функций для константных объектов (некоторые компиляторы только выдают предупреждение). Это жесткое ограничение, поскольку клиенты объекта, возможно, захотят использовать с этим объектом различные «get»-функции. Чтобы предоставить клиентам такую возможность, программист может объявить константные элементы-функции; только эти функции могут оперировать с константными объектами. Разумеется, константные функции-элементы не могут модифицировать объект.

Распространенная ошибка программирования 17.1

Определение с модификатором **const** элемента-функции, которая изменяет элемент данных объекта.

Распространенная ошибка программирования 17.2

Определение с модификатором **const** элемента-функции, которая вызывает не-константную функцию-элемент.

Распространенная ошибка программирования 17.3

Вызов не-константной функции-элемента для константного объекта.

Распространенная ошибка программирования 17.4

Попытка изменения константного объекта.

Функция специфицируется в качестве константной как в ее объявлении, так и в определении путем вставки ключевого слова **const** после списка параметров функции (т.е. в случае определения функции — перед левой фигурной скобкой, с которой начинается ее тело). Например, функция-элемент

```
int getValue() const {return privateDataMember};
```

которая просто возвращает значение одного из элементов данных объекта, объявляется константной. Если константная функция-элемент определяется вне определения класса, то как объявление этой функции, так и ее определение должны включать модификатор **const**.

Общее методическое замечание 17.2

Константная функция-элемент может быть перегружена своей не-константной версией. Компилятор автоматически выбирает, какая из перегруженных функций должна быть вызвана, исходя из того, был ли объект объявлен как **const**.

Здесь возникает интересная проблема в связи с конструкторами и деструкторами, каждый из которых, несомненно, изменяет объекты. Для конструкторов и деструкторов константных объектов объявлений с модификатором **const** не требуется. Конструктору должно быть разрешено изменять объект, с тем чтобы этот объект мог быть надлежащим образом инициализирован. Деструктор должен иметь возможность выполнять свою заключительную «приборку» до разрушения объекта.

В программе на рис. 17.1 создается константный экземпляр объекта класса **Time** и делается попытка его изменения при помощи не-константных элементов-функций **setHour**, **setMinute** и **setSecond**. В окне вывода показаны предупреждения, генерируемые компилятором Borland C++. Компилятор был настроен таким образом, чтобы в случае выдачи любых предупреждений компиляция не проводилась.

```
// TIME5.H
// Объявление класса Time.
// Элементы-функции определены в TIME5.CPP

#ifndef TIME5_H
#define TIME5_H

class Time {
public:
   Time(int = 0, int = 0, int = 0);  // конструктор по умолчанию

   // set-функции
   void setTime(int, int, int);  // устанавливает время
   void setHour(int);       // устанавливает элемент hour
   void setMinute(int);     // устанавливает элемент minute
   void setSecond(int);     // устанавливает элемент second

   // get-функции (обычно объявляются константными)
   int getHour() const;     // возвращает hour
   int getMinute() const;   // возвращает minute
   int getSecond() const;   // возвращает second
```

Рис. 17.1. Использование класса **Time** с константными объектами и константными элементами-функциями (часть 1 из 3)

```
   // print-функции (обычно объявляются константными)
   void printMilitary() const; // выводит время в военном формате
   void printStandard() const; // выводит время в стандартном
                               // формате
private:
   int hour;       // 0 - 23
   int minute;     // 0 - 59
   int second;     // 0 - 59
};

#endif

// TIME5.CPP
// Определения элементов-функций класса Time.
#include "time5.h"
#include <iostream.h>

// Конструктор для инициализации закрытых данных.
// Значения по умолчанию равны 0 (см. определение класса).
Time::Time(int hr, int min, int sec)
{
   hour = ( hr >= 0 && hr < 24) ? hr : 0;
   minute = ( min >= 0 && min < 60) ? min : 0;
   second = ( sec >= 0 && sec < 60) ? sec : 0;
}

// Устанавливает значения элементов hour, minute и second.
void Time::setTime(int h, int m, int s)
{
   hour = ( h >= 0 && h < 24) ? h : 0;
   minute = ( m >= 0 && m < 60) ? m : 0;
   second = ( s >= 0 && s < 60) ? s : 0;
}

// Устанавливает значение элемента hour
void Time::setHour(int h)
   { hour = ( h >= 0 && h < 24) ? h ; 0; }

// Устанавливает значение элемента minute
void Time::setTime(int m)
   { minute = (m >= 0 && M < 60) ? m : 0;  }

// Устанавливает значение элемента second
void Time::setSecond(int s);
   { second = ( s >= 0 && s < 60) ? s : 0;  }

// Получает значение hour
int Time::getHour() const  { return hour;  }

// Получает значение minute
int Time::getMinute() const  {  return minute;  }

// Получает значение second
int Time::getSecond() const  {  return second;  }
```

Рис. 17.1. Использование класса **Time** с константными объектами и константными элементами-функциями (часть 2 из 3)

```
// Отображает время в военном формате: HH:MM:SS
void Time::printMilitary() const
{
   cout << ( hour < 10 ? "0" : "") << hour << ":"
        << ( minute < 10 ? "0" : "") << minute << ":"
        << ( second < 10 ? "0" : "") << second;
}

// Отображает время в стандартном формате: HH:MM:SS AM (или PM)
void Time::printStandard() const
{
   cout << (( hour == 0 || hour == 12) ? 12 : hour % 12) << ":"
        << (minute < 10 ? "0" : "") << minute << ":"
        << (second < 10 ? "0" : "") << second
        << (hour < 12 ? " AM" : " PM");
}

// FIG17_1.CPP
// Попытка доступа к константному объекту
// с помощью не-константных элементов-функций.
#include <iostream.h>
#include "time5.h"

main()
{
   const Time t(19, 33, 52);  // константный объект

   t.setHour(12);     // ОШИБКА: не-константная элемент-функция
   t.setMinute(20);   // ОШИБКА: не-константная элемент-функция
   t.setSecond(39);   // ОШИБКА: не-константная элемент-функция

   return 0;
}
```

```
Compiling FIG17_1.CPP:
Warning FIG17_1.CPP: Non-const function
Time::setHour(int) called for const object
Warning FIG17_1.CPP: Non-const function
Time::setMinute(int) called for const object
Warning FIG17_1.CPP: Non-const function
Time::setSecond(int) called for const object
```

Рис. 17.1. Использование класса **Time** с константными объектами и константными элементами-функциями (часть 3 из 3)

Хороший стиль программирования 17.1

Объявляйте константными все элементы-функции, предназначенные для использования с константными объектами.

Константный объект нельзя изменить с помощью присваивания, поэтому он должен быть инициализирован. Если элемент данных является константным объектом, то для того, чтобы конструктор мог установить начальные значения для этого объекта, должен использоваться *инициализатор элемента*. На рис. 17.2 показаны примеры использования синтаксиса инициализатора

элемента для константного элемента данных **increment** класса **Increment**. Конструктор класса **Increment** модифицирован следующим образом:

```
Increment::Increment(int c, int i)
   : increment(i)
{ count = c; }
```

```
// FIG17_2.CPP
// Использование инициализатора элемента для инициализации
// константы встроенного типа данных.

#include <iostream.h>

class Increment  {
public:
   Increment(int c = 0, int i = 1);
   void addIncrement()  { count += increment;  }
   void print() const;
private:
   int count;
   const int increment;  // константный элемент данных
};

// Конструктор класса Increment
Increment::Increment(int c, int i)
   : increment(i)  // Инициализатор элементата данных
{  count = c; }

// Выводит данные
void Increment::print() const
{
   cout << "count = " << count
        << ", increment = " << increment << endl;
}

main()
{
   Increment value(10, 5);

   cout << "Before incrementing: ";
   value.print();

   for (int j = 1; j <= 3; j++) {
      value.addIncrement();
      cout << "After increment " << j << ": " ;
      value.print();
   }

   return 0;
}
```

```
Before incrementing:  count = 10,  increment = 5
After increment 1:  count = 15,  increment = 5
After increment 2:  count = 20,  increment = 5
After increment 3:  count = 25,  increment = 5
```

Рис. 17.2. Использование инициализатора элемента для инициализации константы встроенного типа данных

Запись **:increment(i)** вызывает инициализацию элемента **increment** значением **i**. В случае необходимости нескольких инициализаторов элементов просто включите их в разделенный запятыми список после двоеточия.

Рис. 17.3 иллюстрирует ошибки компилятора при трансляции программы, которая пытается инициализировать **increment** с помощью операции присваивания, а не инициализатора элемента.

```
// FIG17_3.CPP
// Попытка инициализации константы
// встроенного типа данных с помощью присваивания.
#include <iostream.h>

class Increment {
public:
   Increment(int c = 0, int i = 1);
   void addIncrement()  { count += increment; }
   void print() const;
private:
   int count;
   const int increment;
};

// Конструктор класса Increment
Increment::Increment(int c, int i)
{                  // Константный элемент данных 'increment'
                   // не инициализирован
   count = c;
   increment = i; // ОШИБКА: Невозможно изменить константный
                   // объект
}

// Выводит данные
void Increment::print() const
{
   cout << "count = " << count
        << ", increment = " << increment << '\n' ;
}

main()
{
   Increment value(10, 5);

   cout << "Before incrementing: " ;
   value.print();

   for (int j = 1; j <= 3; j++) {
      value.addIncrement();
      cout << "After increment " << j << " : " ;
      value.print();
   }

   return 0;
}
```

Рис. 17.3. Ошибочная попытка инициализации константы встроенного типа данных с помощью присваивания (часть 1 из 2)

```
Compiling FIG17_3.CPP:
Warning FIG17_3.CPP 18: Constant member 'increment' is
   not initialized
Ошибка FIG17_3.CPP 20: Cannot modify a const object
Warning FIG17_3.CPP 21: Parameter 'i' is never used
```

Рис. 17.3. *Ошибочная попытка инициализации константы встроенного типа данных с помощью присваивания (часть 2 из 2)*

Распространенная ошибка программирования 17.5

Не предусматривают инициализатор элемента для константного объекта - элемента данных.

Общее методическое замечание 17.3

Как константные объекты, так и константные «переменные» необходимо инициализировать, используя синтаксис инициализатора элемента. Присваивания в этом случае не допускаются.

17.3. Композиция: классы в качестве элементов других классов

Объекту класса **AlarmClock** (будильник) необходимо знать о времени, когда он должен зазвонить, так почему бы не включить объект **Time** в качестве элемента объекта **AlarmClock**? Такое включение называется *композицией*. Класс может иметь в качестве элементов другие классы.

Общее методическое замечание 17.4

Одной из форм повторного использования программного кода является композиция, когда класс содержит в качестве элементов объекты других классов.

При вхождении объекта в область действия его конструктор вызывается автоматически, поэтому нам необходимо специфицировать способ передачи аргументов в конструкторы объектов — элементов данных. Элементы данных — объекты создаются до создания объектов включающего их класса.

Общее методическое замечание 17.5

Если объект содержит несколько объектов в качестве элементов данных, эти элементы данных – объекты создаются в том порядке, в котором они были определены. Тем не менее, не пишите кода, который зависит от специфического порядка выполнения конструкторов этих объектов.

Программа на рис. 17.4 использует классы **Employee** и **Date** для демонстрации объектов в качестве элементов данных других объектов. Класс **Employee** содержит закрытые элементы данных **lastName**, **firstName**, **birthDate** и **hireDate**. Элементы **birthDate** и **hireDate** являются объектами класса **Date**, который содержит закрытые элементы данных **month**, **day** и **year**. Программа создает экземпляр объекта **Employee** и инициализирует, а затем выводит его

элементы данных. Обратите внимание на синтаксис заголовка в определении конструктора **Employee**:

```
Employee::Employee(char *fname, char *lname,
                   int bmonth, int bday, int byear,
                   int hmonth, int hday, int hyear)
   : birthDate(bmonth, bday, byear),
     hireDate(hmonth, hday, hyear)
```

Конструктор принимает восемь параметров (**fname**, **lname**, **bmonth**, **bday**, **byear**, **hmonth**, **hday** и **hyear**). Двоеточие в заголовке отделяет инициализаторы элементов от списка параметров. Инициализаторы элементов специфицируют параметры **Employee**, которые передаются в конструкторы элементов данных — объектов. Таким образом, **bmonth**, **bday** и **byear** передаются в конструктор **birthDate** и **hmonth**, **hday** и **hyear** передаются в конструктор **hireDate**. Если инициализаторов несколько, они отделяются друг от друга запятыми.

```
// DATE1.H
// Объявление класса Date.
// Элементы-функции определены в DATE1.CPP
#ifndef DATE1_H
#define DATE1_H

class Date  {
public:
   Date(int = 1, int = 1, int = 1990); // конструктор
                                       // по умолчанию
   void print() const;  // выводит дату в формате  месяц/день/год

private:
   int month;  // 1- 12
   int day;    // 1 - 31 в зависимости от месяца
   int year;   // любой год

   // сервисная функция для проверки корректности дня
   // для данных месяца и года
   int checkDay(int);
};

#endif

// DATE1.CPP
// Определения элементов-функций для класса Date.

#include <iostream.h>
#include "date1.h"

// Конструктор: подтверждает корректность значения month;
// вызывает сервисную функцию checkDay для подтверждения
// корректности значения для day.
Date:Date(int mn, int dy, int yr)
{
   month = ( mn > 0 && mn <= 12) ? mn : 1;
                                  // проверяет корректность
   year = yr;                     // также могла бы проверять
   day = checkDay(dy);            // проверяет корректность
```

Рис. 17.4. Использование инициализаторов для объектов — элементов данных (часть 1 из 3)

```
   cout << "Date object constructor for date " ;
   print();
   cout << endl;
}

// Сервисная функция для подтверждения корректности значения
// day, исходя из значений month и year.
int Date:checkDay(int testDay)
{
   static int daysPerMonth[13] = {0, 31, 28, 31, 30,
                                  31, 30, 31, 31, 30,
                                  31, 30, 31};

   if ( month != 2) {

      if ( testDay > 0 && testDay <= daysPerMonth[month])
         return testDay;

   }
   else { // Февраль: проверяет возможность високосного года
      int days = (year % 400 == 0 ||
                  (year % 4 == 0 && year % 100 != 0) ? 29 : 28);

      if ( testDay > 0 && testDay <= days)
         return testDay;
   }

   cout << "Day" << testDay << " invalid. Set to day 1.\n";
   return 1;  // оставляет объект в целостном состоянии в случае
              // некорректого значения
}

// Выводит объект Date в формате  месяц/день/год
void Date::print() const
   { cout << month << '/' << day << '/' << year; }

// EMPLY1.H
// Объявление класса Employee.
// Элементы-функции определены в EMPLY1.CPP
#ifndef EMPLY1_H
#define EMPLY1_H

#include "date1.h"

class Employee {
public:
   Employee(char *, char *, int, int, int, int, int, int);
   void print() const;
private:
   char lastName[25];
   char firstName[25];
   Date birthDate;
   Date hireDate;
};

#endif
```

Рис. 17.4. Использование инициализаторов для объектов — элементов данных (часть 2 из 3)

```
// EMPLY1.CPP
// Определения элементов-функций класса Employee.
#include <iostream.h>
#include <string.h>
#include "emply1.h"
#include "date1.h"

Employee::Employee(char * fname, char *lname,
                   int bmonth, int bday, int byear,
                   int hmonth, int hday, int hyear)
   : birthDate(bmonth, bday, byear),
     hireDate(hmonth, hday, hyear)
{
   strncpy(firstName, fname, 24);
   firstName[24] = '\0 ';
   strncpy(lastName, lname, 24);
   lastName[24] = '\0 ';
   cout << "Employee object constructor: "
        << firstName << ' ' << lastName << endl;
}

void Employee::print() const
{
   cout << lastName << ", " << firstName << "\nHired: " ;
   hireDate.print();
   cout << " Birthday: " ;
   birthDate.print();
   cout << endl;
}

// FIG17_4.CPP
// Демонстрация объекта с элементом данных-объектом.
#include <iostream.h>
#include "emply1.h"

main()
{
   Employee e("Bob", "Jones", 7, 24, 49, 3, 12, 88);

   cout << endl;
   e.print();

   cout << "\nTest Date constructor with invalid values:\n" ;
   Date d(14, 35, 94);  // некорректные значения для объекта Date

   return 0;
}
```

```
Date object constructor for date 7/24/49
Date object constructor for date 3/12/88
Employee:  Bob Jones
Jones, Bob
Hired: 3/12/88  Birthday: 7/24/49
Test Date constructor with invalid values:
Day 35 invalid. Set to day 1.
Date object constructor for date 1/1/94
```

Рис. 17.4. Использование инициализаторов для объектов — элементов данных (часть 3 из 3)

Элемент данных — объект не обязательно инициализировать с помощью инициализатора элемента. Если инициализатор элемента не задан, автоматически будет вызван конструктор по умолчанию. Значения, установленные конструктором по умолчанию (если вообще такие имеются), могут быть в последствии изменены при помощи set-функций.

Распространенная ошибка программирования 17.6

Не объявлен конструктор по умолчанию для объекта — элемента данных, когда для него нет инициализатора элемента. Компилятор в этом случае генерирует ошибку.

Совет по повышению эффективности 17.1

Явно инициализируйте объекты — элементы данных посредством инициализаторов элементов. Это устраняет накладные расходы, связанные с двойной инициализацией элементов-объектов, — первый раз при вызове конструктора по умолчанию для объекта — элемента данных и снова при инициализации элемента с помощью set-функций.

17.4. Дружественные функции и дружественные классы

Дружественная функция класса определена вне области действия этого класса, однако она имеет право доступа к его закрытым (и, как мы увидим в главе 19, «Наследование», защищенным) элементам. Функция или целый класс могут быть объявлены в качестве «друзей» иного класса.

Чтобы объявить функцию в качестве друга некоторого класса, поместите перед прототипом этой функции в определении класса ключевое слово **friend**. Чтобы объявить класс **ClassTwo** в качестве друга класса **ClassOne**, поместите объявление вида

```
friend ClassTwo;
```

в качестве элемента класса **ClassOne**.

Общее методическое замечание 17.6

Понятия типа доступа — **private**, **protected** и **public** — не имеют смысла для объявлений дружественных функций и классов, поэтому эти объявления могут размещаться в любом месте определения класса.

Хороший стиль программирования 17.2

Помещайте все объявления дружественных функций и классов в начале определения класса сразу после его заголовка и не помещайте перед этими объявлениями никаких спецификаторов доступа.

Дружба даруется, но не может быть навязана, т.е. для того, чтобы класс B был другом класса A, в классе A должно быть объявлено, что класс B является его другом. Кроме того, *понятие дружественности не является ни симметричным*, ни транзитивным, т.е. из того, что класс A является другом класса B, а класс B является другом класса C, вы не можете заключить ни того, что класс B является другом класса A, ни того, что класс C является другом класса B, а также того, что класс A является другом класса C.

Общее методическое замечание 17.7

Некоторые программисты считают, что «дружественность» разрушает сокрытие информации и ослабляет значимость объектно-ориентированного подхода к проекту.

В программе на рис. 17.5 демонстрируется объявление и использование дружественной функции setX для установки закрытого элемента данных x класса Count. Обратите внимание, что объявление friend появляется (по соглашению) первым в объявлении класса, даже до объявления открытых элементов-функций. В программе на рис. 17.6 демонстрируются сообщения компилятора, возникающие при вызове не-дружественной функции cannotSetX для изменения закрытого элемента данных x.

```
// FIG17_5.CPP
// Друзья класса могут обращаться к его закрытым элементам.
#include <iostream.h>

// Модифицированный класс Count
class Count {
   friend void setX ( Count &, int);  // объявление дружественной
                                      // функции
public:
   Count() { x = 0; }                 // конструктор
   void print() const { cout << x << endl; }   // вывод
private:
   int x;  // элемент данных
};

// Может изменять закрытые данные Count, поскольку
// setX объявлена в качестве дружественной функции класса Count
void setX(Count &c, int val)
{
   c.x = val;  / допустимо: setX является другом Count
}

main()
{
   Count object;

   cout << "object.x after instantiation: " ;
   object.print();
   cout << "object.x after call to setX friend function: " ;
   setX(object, 8); // установка x с помощью дружественной
                    // функции
   object.print();

   return 0;
}
```

```
object.x after instantiation:  0
object.x after call to setX friend function:  8
```

Рис. 17.5. Друзья класса могут обращаться к его закрытым элементам

```
// FIG17_6.CPP
// Функции, не являющиеся элементами или друзьями класса,
// не могут обращаться к его закрытым элементам.
#include <iostream.h>

// Модифицированный класс Count
class Count {
public:
   Count()  { x = 0; }   // конструктор
   void print() const  { cout << x << '\n' ; }  // вывод
private:
   int x;  // элемент данных
};

// Функция пытается изменить закрытые данные Count,
// но не может этого сделать, поскольку она не является
// другом класса Count.
void cannotSetX(Count &c, int val)
{
   c.x = val;  // ОШИБКА: 'Count::x' недоступен
}

main()
{
   Count object;

   cannotSetX( object, 3);  // cannotSetX не является другом

   return 0;
}
```

```
Compiling FIG17_6.CPP:
Error FIG17_6.CPP  17: 'Count::x' is not accessible
Warning FIG17_6.CPP  18: Parameter 'c' is never used
Warning FIG17_6.CPP  18: Parameter 'val' is never used
```

Рис. 17.6. Функции, не являющиеся элементами или друзьями класса, не могут обращаться к его закрытым элементам

Можно определять в качестве друзей класса перегруженные функции. Каждая перегруженная функция, которую собираются сделать дружественной, должна быть явно объявлена в определении класса в качестве друга.

17.5. Указатель this

Когда элемент-функция ссылается на другой элемент класса в некотором объекте этого класса, каким образом C++ гарантирует, что происходит ссылка на надлежащий объект? Ответ состоит в том, что каждый объект поддерживает указатель на самого себя — называемый *указателем* ***this*** — который является неявным аргументом во всех ссылках на элементы внутри этого объекта. Указатель **this** может также использоваться явно. Каждый объект может определить свой собственный адрес, используя ключевое слово **this**.

Указатель **this** неявно используется для ссылок как на элементы данных, так и на элементы-функции объекта. Тип указателя **this** зависит от типа объекта, а также от того, объявлена ли константной функция-элемент, в которой используется указатель **this**. В не-константной функции-элементе класса **Employee** указатель **this** имеет тип **Employee * const** (константный указатель на объект **Employee**). В константной функции-элементе класса **Employee** указатель **this** имеет тип **Employee const * const** (константный указатель на константный объект **Employee**).

Сейчас мы покажем простой пример явного использования указателя **this**; позже мы продемонстрируем некоторые важные и тонкие примеры использования **this**-указателя. Каждая функция-элемент имеет доступ к указателю **this** на объект, для которого она вызывается.

На рис. 17.7 демонстрируется явное использование указателя **this**, посредством которого функция-элемент класса **Test** выводит закрытый элемент данных **x** объекта **Test**. Программа использует как операцию-стрелку (**->**) с указателем **this**, так и операцию-точку (**.**) с разыменованным указателем **this**.

```
// FIG17_7.CPP
// Использование указателя this для ссылки на элементы объекта.
#include <iostream.h>

class Test {
public:
   Test(int = 0);
   void print() const;
private:
   int x;
};

Test::Test (int a ) { x = a; }  // конструктор

void Test::print() const
{
   cout << "       x = " << x
        << "\n this->x = " << this->x
        << "\n(*this).x = " << (*this).x << '\n' ;
}

main()
{
   Test a(12);

   a.print();

   return 0;
}
```

```
       x = 12
  this->x = 12
(*this).x = 12
```

Рис. 17.7. Использование указателя **this**

Совет по повышению эффективности 17.2

Из-за соображений экономии памяти для класса существует только одна копия каждого элемента-функции, которая вызывается всеми объектами этого класса. С другой стороны, каждый объект имеет собственную копию элементов данных класса.

Обратите внимание на круглые скобки вокруг ***this** при использовании указателя **this** с операцией выбора элемента (.). Круглые скобки требуются, поскольку операция-точка имеет более высокий приоритет, чем операция *. Без круглых скобок выражение

```
*this.x
```

оценивалось бы так, как если бы скобки были расставлены следующим образом:

```
*(this.x)
```

Компилятор C++ обработал бы это выражение как синтаксическую ошибку, поскольку операция выбора элемента не может применяться к указателю.

Распространенная ошибка программирования 17.7

Попытка применить операцию выбора элемента (.) к указателю на объект (операция выбора элемента может быть использована только с объектом или ссылкой на объект).

Одним интересным применением указателя **this** является недопущение присваивания объекта себе самому. Как мы увидим в главе 18, «Перегрузка операций», присваивание объекта себе самому может вызвать серьезные ошибки, если объекты содержат указатели на память, динамически распределенную операцией **new**.

Рис. 17.8 иллюстрирует возвращение ссылки на объект **Time**, что дает возможность конкатенации вызовов функций-элементов класса **Time**. Каждая из функций **setTime**, **setHour**, **setMinute** и **setSecond** возвращает ссылку на **Time** со значением ***this** .

Почему работает метод, основанный на возвращении ***this** в качестве ссылки? Операция-точка (.) ассоциируется слева направо, поэтому в выражении

```
t.setHour(18).setMinute(30).setSecond(22);
```

сначала оценивается **t.setHour(18)**, затем возвращается ссылка на объект **t** в качестве значения вызова этой функции. Оставшееся выражение затем интерпретируется как

```
t.setMinute(30).setSecond(22);
```

Выполняется вызов **t.setMinute(30)** и возвращается эквивалент **t**. Оставшееся выражение интерпретируется как

```
t.setSecond(22);
```

Обратите внимание, что в вызовах

```
t.setTime(20, 20, 20).printStandard();
```

также используется свойство конкатенации. Эти вызовы должны появляться в этом выражении именно в таком порядке, поскольку **printStandard**, согласно своему определению в классе, не возвращает ссылки на **t**. Помещение в предыдущем операторе вызова **printStandard** перед вызовом **setTime** приводит к синтаксической ошибке.

```
// TIME6.H
// Объявление класса Time.
// Элементы-функции определены в TIME6.CPP

#ifndef TIME6_H
#define TIME6_H

class Time {
public:
   Time ( int = 0, int = 0, int = 0); // конструктор по умолчанию

   // set-функции
   Time &setTime(int, int, int);  // устанавливает hour, minute,
                                  // second
   Time &setHour(int);     // устанавливает hour
   Time &setMinute(int);   // устанавливает minute
   Time &setSecond(int);   // устанавливает second

   // get-функции (обычно объявляются константными)
   int getHour() const;     // возвращает hour
   int getMinute() const;   // возвращает minute
   int getSecond() const;   // возвращает second

   // print-функции (обычно объявляются константными)
   void printMilitary() const;  // выводит время
                                // в военном формате
   void printStandard() const;  // выводит время
                                // в стандартном формате
private:
   int hour;      // 0 - 23
   int minute;    // 0 - 59
   int second;    // 0 - 59
};

#endif

// TIME6.CPP
// Определения элементов-функций класса Time.

#include "time6.h"
#include <iostream.h>

// Конструктор для инициализации закрытых данных.
// Значения по умолчанию равны 0 (см. определение класса).
Time::Time(int hr, int min, int sec)
{
   hour = ( hr >= 0 &&hr < 24) ? hr : 0;
   minute = ( min >= ) && min < 60) ? min : 0;
   second = ( sec >= 0 && sec < 60) ? sec : 0;
}

// Устанавливает значения элементов hour, minute и second.
Time &Time::setTime(int h, int m, int s)
{
   hour = ( h >= 0 && h < 24) ? h : 0;
```

Рис. 17.8. Формирование цепочки вызовов элементов-функций (часть 1 из 3)

```
   minute = ( m >= 0 && m < 60) ? m : 0;
   second = ( s >= 0 && s < 60) ? s : 0;
   return *this; // дает возможность формирования цепочки вызовов
}
// Устанавливает значение элемента hour
Time &Time::setHour(int h)
{
   hour = ( h >= 0 && h < 24) ? h : 0;
   return *this; // дает возможность формирования цепочки вызовов
}
// Устанавливает значение элемента minute
Time &Time::setMinute(int m)
{
   minute = ( m >= 0 && m < 60) ? m : 0;
   return *this; // дает возможность формирования цепочки вызовов
}
// Устанавливает значение элемента second
Time &Time::setSecond(int s)
{
   second = ( s >= 0 && s < 60) ? s : 0;
   return *this; // дает возможность формирования цепочки вызовов
}
// Получает значение hour
int Time::getHour() const { return hour; }

// Получает значение minute
int Time::getMinute() const { return minute; }

// Получает значение second
int Time::getSecond() const { return second; }

// Отображает время в военном формате: HH:MM:SS
void Time::printMilitary() const
{
   cout << (hour < 10 ? "0" : "") << hour << ":"
        << (minute < 10 ? "0" : "") << minute << ":"
        << (second < 10 ? "0" : "") << second;
}
// Отображает время в стандартном формате: HH:MM:SS AM (или PM)
void Time::printStandard() const
{
   cout << ((hour == 0 || hour == 12) ? 12 : hour % 12) << ":"
        << (minute < 10 ? "0" : "") << minute << ":"
        << (second < 10 ? "0" : "") << second
        << (hour < 12 ? " AM" : " PM");
}

// FIG17_8.CPP
// Формирование цепочки вызовов элементов-функций
// при помощи указателя this
#include <iostream.h>
#include "time6.h"

main()
{
```

Рис. 17.8. Формирование цепочки вызовов элементов-функций (часть 2 из 3)

```
   Time t;

   t.setHour(18).setMinute(30).setSecond(22);
   cout << "Military time: ";
   t.printMilitary();
   cout << "\nStandard time: ";
   t.printStandard();

   cout << "\n\nNew standard time: ";
   t.setTime(20, 20, 20).printStandard();
   cout << endl;

   return 0;
}
```

```
Military time:  18:30:22
Standard time:  6:30:22 PM

New standard time:  8:20:20 PM
```

Рис. 17.8. Формирование цепочки вызовов элементов-функций (часть 3 из 3)

17.6. Динамическое распределение памяти с помощью операций new и delete

Операции **new** и **delete** предоставляют программисту более изящный способ управления динамическим распределением памяти по сравнению с вызовами функций **malloc** и **free** в языке C. Рассмотрим следующее объявление кода:

```
TypeName *typeNamePtr;
```

В ANSI C для динамического создания объекта типа **TypeName** вы бы написали

```
TypeNamePtr = malloc(sizeof(TypeName));
```

Здесь требуется обращение к функции **malloc** и явное применение операции **sizeof**. В версиях языка C, предшествующих ANSI C, вы должны были бы также преобразовать указатель, возвращенный **malloc**, с помощью операции приведения типа **(TypeName *)**. Функция **malloc** не предусматривает какого-либо метода инициализации блока выделенной памяти. В C++ вы просто пишете

```
TypeNamePtr = new TypeName;
```

Операция **new** автоматически создает объект надлежащего размера, вызывает для объекта конструктор (если таковой доступен) и возвращает указатель соответствующего типа. Если **new** не может найти свободной памяти, она возвращает указатель **0**. В C++ для освобождения памяти, выделенной этому объекту, используется операция **delete**:

```
delete typeNamePtr;
```

C++ допускает спецификацию инициализатора для вновь созданного объекта, например:

```
float *thingPtr = new float (3.14159);
```

который инициализирует вновь созданный объект типа **float** значением **3.14159**.

Массив может быть создан и присвоен **int *chessBoardPtr** следующим образом:

```
chessBoardPtr = new int[8][8];
```

Этот массив может быть удален с помощью оператора

```
delete [] chessBoardPtr;
```

Как мы увидим, использование **new** и **delete** вместо **malloc** и **free** имеет также и другие преимущества. В частности, операция **new** автоматически вызывает конструктор, а **delete** автоматически вызывает деструктор класса.

Распространенная ошибка программирования 17.8

Смешивание динамического распределения памяти в стиле **new** и **delete** с вызовами **malloc** и **free**. Память, выделенная функцией **malloc**, не может быть освобождена с помощью **delete**; объекты, созданные с помощью **new**, не могут быть удалены функцией **free**.

Хороший стиль программирования 17.3

Хотя в программах на C++ может применяться как память, выделяемая с помощью **malloc** и освобождаемая с помощью **free**, так и объекты, создаваемые с помощью **new** и удаляемые с помощью **delete**, все же лучше использовать только **new** и **delete**.

17.7. Статические элементы класса

Обычно каждый объект класса имеет собственную копию всех элементов данных класса. Иногда это становится расточительным. В некоторых случаях все элементы класса должны разделять только одну копию конкретного элемента данных. По этой и другим причинам используются статические элементы данных. Статический элемент данных представляет собой «общеклассовую» информацию. Объявление статического элемента начинается с ключевого слова **static**.

Совет по повышению эффективности 17.3

Используйте статические элементы данных с целью экономии памяти, если достаточно только одной копии данных.

Хотя может показаться, что статические элементы данных подобны глобальным переменным, они имеют область действия класса. Статические элементы могут быть открытыми, закрытыми или защищенными. Статические элементы данных должны быть инициализированы в области действия файла. К открытым статическим элементам класса можно обращаться посредством любого объекта этого класса, либо посредством имени класса с использованием операции разрешения области действия. К закрытым и защищенным статическим элементам класса обращение должно происходить посредством открытых элементов-функций класса. Статические элементы класса существуют, даже если не существует ни одного объекта этого класса. Для обращения к открытому статическому элементу класса в случае, когда не существует ни одного объекта этого класса, просто поместите перед именем элемента имя класса с операцией разрешения области действия.

Для обращения к закрытому или защищенному статическому элементу класса в случае, когда не существует ни одного объекта этого класса, должна быть предусмотрена открытая статическая функция-элемент, при вызове которой ее имени должны предшествовать имя класса и операция разрешения области действия.

В программе на рис. 17.9 демонстрируется использование закрытого статического элемента данных и открытой статической функции-элемента. Элемент данных **count** инициализирован нулем в области действия файла с помощью оператора

```
int Employee::count = 0;
```

Элемент данных **count** содержит общее число уже созданных объектов класса **Employee**. Если объекты класса **Employee** существуют, то на элемент **count** может ссылаться любая функция-элемент объекта класса **Employee** — в нашем примере, на **count** ссылаются как конструктор, так и деструктор. Если не существует ни одного объекта класса **Employee**, то на элемент **count** все же можно ссылаться, однако только путем следующего вызова статической функции-элемента **getCount**:

```
Employee::getCount()
```

В этом примере функция **getCount** используется для определения числа созданных к настоящему моменту объектов **Employee**. Обратите внимание, что в случае отсутствия представителей класса применяется показанный выше вызов функции. Однако, если представители класса существуют, функция **getCount** может быть вызвана через один из них:

```
e1Ptr->getCount()
```

```
// EMPLOY1.H
// Класс Employee
#ifndef EMPLOY1_H
#define EMPLOY1_H

class Employee {
public:
   Emplouee(const char*, const char*);  // конструктор
   ~Employee();  // деструктор
   char *getFirstName() const;  // возвращает имя
   char *getLastName() const;   // возвращает фамилию

   // статическая элемент-функция
   static int getCount(); // возвращает число созданных
                          // экземпляров объектов
private:
   char *firstName;
   char *lastName;

   // статический элемент данных
   static int count;  // количество созданных
                      // экземпляров объектов
};

#endif
```

Рис. 17.9. Использование статического элемента данных для хранения числа объектов класса (часть 1 из 3)

```
// EMPLOY1.CPP
// Определения элементов-функций класса Employee
#include <iostream.h>
#include <string.h>
#include <assert.h>
#include "employ1.h"

// Инициализирует статический элемент данных
int Employee::count = 0;

// Определяет статическую элемент-функцию, которая
// возвращает количество созданных экземпляров объектов Employee.
int Employee::getCount() { return count; }

// Конструктор динамически распределяет память для
// имени и фамилии и использует функцию strcpy для копирования
// имени и фамилии в объект
Employee::Employee(const char *first, const char *last)
{
   firstName = new char[ strlen(first) + 1 ];
   assert(firstName != 0);  // убеждается, что память выделена
   strcpy(firstName, first);

   lastName = new char[ starlen(last) + 1 ];
   assert(lastName != 0);   // убеждается, что память выделена
   strcpy(lastName, last);

   ++count;  // увеличивает статический элемент данных count
   cout < "Employee constructor for " << firstName
        << ' ' << lastName << " called.\n" ;
}

// Деструктор освобождает динамически распределенную память
Employee::~Employee()
{
   cout << "~Emplouee() called for " << firstName
        << ' ' << lastName << endl;
   delete [] firstName;  // возвращает обратно память
   delete [] lastName;   // возвращает обратно память
   —count;  // уменьшает статический элемент данных count
}

// Возвращает имя служащего
char *Emplouee::getFirstName() const
{
   char *tempPtr = new char[strlen(firstName) + 1];
   assert ( tempPtr != 0);  // убеждается, что память выделена
   strcpy ( tempPtr, firstName);
   return tempPtr;
}

// Возвращает фамилию служащего
char *Employee::getLastName() const
{
   char *tempPtr = new char[strlen(lastName) + 1];
```

Рис. 17.9. Использование статического элемента данных для хранения числа объектов класса (часть 2 из 3)

```
   assert(tempPtr != 0);   // убеждается, что память выделена
   strcpy(tempPtr, lastName);
   return tempPtr;
}

// FIG17_9.CPP
// Программа для тестирования класса Employee
#include <iostream.h>
#include "employ1.h"

main()
{
   cout << "Number of employees before instantiation is "
        << Employee::getCount() << endl;  // использует имя клас-
са

   Employee *e1Ptr = new Employee('Susan", "Baker");
   Employee *e2Ptr = new Employee("Robert", "Jones");

   cout << "Number of employees after instantiation is"
        << e1Ptr->getCount() << endl;

   cout << "\nEmployee 1: "
        << e1Ptr->getFirstName()
        << " " << e1Ptr->getLastName()
        << "\nEmployee 2: "
        << e2Ptr->getFirstName()
        << " " << e2Ptr->getLastName() << "\n\n";

   delete e1Ptr;  // возвращает обратно память
   delete e2Ptr;  // возвращает обратно память

   cout << "Number of employees after deletion is "
        << Employee::getCount() << endl;

   return 0;
}
```

```
Number of employees before instantiation is 0
Employee constructor for Susan Baker called.
Employee constructor for Robert Jones called.
Number of employees after instantiation is 2

Employee 1: Susan Baker
Employee 2: Robert Jones

~Employee() called for Susan Baker
~Employee() called for Robert Jones
Number of employees after deletion is 0
```

Рис. 17.9. Использование статического элемента данных для хранения числа объектов класса (часть 3 из 3)

Функция-элемент может быть объявлена статической, если она не обращается к не-статическим элементам класса. В отличие от не-статических элементов-функций, статическая функция-элемент не имеет указателя **this**, по-

скольку статические элементы данных и статические элементы-функции существуют независимо от объектов класса.

Распространенная ошибка программирования 17.9

Обращение к указателю **this** внутри статической функции-элемента.

Распространенная ошибка программирования 17.10

Объявление статической функции-элемента с модификатором **const**.

Общее методическое замечание 17.8

Статические элементы данных и статические элементы-функции существуют и к ним можно обращаться, даже если не было создано ни одного представителя класса.

17.8. Абстракция данных и сокрытие информации

Классы обычно скрывают подробности своей реализации от своих клиентов (т.е. пользователей). Это называется сокрытием информации. Программист может, например, создать класс стека и скрыть от пользователей его реализацию. Стеки могут быть легко реализованы с помощью массивов или связанных списков. Клиент класса стека не должен знать о его внутреннем устройстве. Что действительно заботит пользователя — так это то, что при помещении элементов данных в стек они будут выбираться в порядке *LIFO* («последним вошел, первым вышел»). Эта концепция носит название *абстракции данных*, а классы C++ определяют *абстрактные типы данных (ADT)*. Хотя пользователи и могут знать детали реализации класса, они не будут писать код, зависящий от этих деталей. Это означает, что данный класс (например, тот, который реализует стек и операции «втолкнуть» и «вытолкнуть») можно заменить его новой версией, не затрагивая при этом остальной системы, до тех пор, пока не изменяется открытый интерфейс класса.

Задачей языка высокого уровня является формирование «точки зрения» на программирование, удобной для программистов. Не существует какой-то одной общепринятой точки зрения — это одна из причин наличия такого большого числа программных языков. Объектно-ориентированное программирование на C++ являет собой еще одну такую точку зрения.

Большинство языков программирования акцентируют действия. В этих языках данные существуют для поддержки действий, которые должны выполнять программы. Во всяком случае, данные «менее интересны», чем действия. Данные «примитивны». Существует только несколько встроенных типов данных, и программистам нелегко создавать свои собственные новые типы.

Такое представление меняется в C++ и объектно-ориентированном подходе к программированию. C++ повышает значимость данных. Основное поле деятельности в C++ — это создание новых типов (т.е. классов) и отражение взаимодействия между объектами этих типов.

Для того, чтобы продвигаться в этом направлении, специалистам в области языков программирования требовалось формализовать некоторые понятия, касающиеся данных. Рассматриваемая нами формализация связана с понятием абстрактного типа данных (ADT). Абстрактные типы данных привлекают

сегодня такое же внимание, как и структурное программирование на протяжении последних двух десятилетий. Абстрактные типы данных не вытесняют структурного программирования. Скорее они создают дополнительный уровень формализации, способствующий дальнейшему усовершенствованию процесса разработки программного обеспечения.

Что же это такое — абстрактный тип данных? Рассмотрим встроенный тип **int**. В голову сразу приходит понятие целого числа в математике, однако **int** на компьютере не совсем то же самое, что обычное целое число. В частности, компьютерные целые числа обычно довольно ограничены по величине. Например, тип **int** на 32-разрядном компьютере ограничен значениями примерно от −2 до +2 миллиардов. Если результат вычисления выходит за пределы этого диапазона, происходит переполнение, и компьютер реагирует некоторым машинно-зависимым образом. С математическими целыми числами этой проблемы не возникает. Таким образом, понятие компьютерного типа **int** на самом деле является только приближением понятия «настоящего» целого числа. Тот же самое верно и для типа **float**.

Даже тип **char** является таким же приближением. Значения типа **char** обычно представляют собой 8-битовые последовательности, которые выглядят совсем не похожими на символы, которые они должны представлять, как, например, прописная буква **Z**, символ нижнего регистра **z**, знак доллара ($), цифра (напр., 5) и т.д. Значения типа **char** на большинстве компьютеров довольно ограниченны в сравнении с множеством реально существующих символов. 7-разрядный набор символов ASCII предусматривает 127 различных символьных значений. Это совершенно недостаточно для представления таких языков, как японский и китайский, для которых требуются тысячи символов.

Дело в том, что даже встроенные типы данных, предоставляемые языками программирования, подобными C++, в действительности являются только приближениями, или моделями, понятий и поведения объектов реального мира. До настоящего времени мы воспринимали тип **int** как нечто само собой *разумеющееся, однако теперь читатель может взглянуть на все по-новому.* Типы, такие как **int**, **float**, **char** и другие, — все они являются примерами так называемых *абстрактных типов данных*. Абстрактные типы данных являются, по существу, способами представления понятий реального мира внутри компьютерной системы, удовлетворяющими определенному уровню точности.

Абстрактный тип данных на самом деле охватывает два понятия, а именно представление данных и операции, которые допустимы над этими данными. Например, понятие **int** в языке C++ определяет операции сложения, вычитания, умножения, деления и взятия по модулю, однако деление на нуль не определено; и выполнению этих разрешенных операций небезразличны параметры компьютера, такие как фиксированный размер слова базовой компьютерной системы. В C++ для реализации абстрактных типов данных программист использует классы.

17.8.1. Пример: абстрактный тип данных — массив

Мы обсуждали массивы в главе 6. Массив мало чем отличается от указателя. Примитивная адресация элементов, встроенная в C, достаточна для выполнения операций над массивами, если программист проявляет осторожность. Существует много операций, которые было бы желательно выполнять над массивами, но которые не встроены в C++. Используя C++, программист может

разработать абстрактный тип данных — массив, более совершенный, чем «сырые» массивы. Класс массива может реализовать множество весьма привлекательных возможностей, например:

- Проверку диапазона индекса.
- Произвольный диапазон индексов.
- Присваивание массивов.
- Сравнение массивов.
- Ввод/вывод для массивов.
- «Знание» массивами своих размеров.

Слабым местом здесь является то, что мы создаем настроенный на заказчика, нестандартный тип данных, который вряд ли будет доступен точно в такой же форме в большинстве реализаций C++. Цель применения C++ и объектно-ориентированного программирования — это более быстрая разработка программ. Для реализации в полной мере потенциала ориентации на объекты в программировании критически важным является работа программистов в направлении крупномасштабной стандартизации и распространения библиотек классов.

C++ включает небольшой набор встроенных типов. Абстрактные типы данных расширяют базовый язык программирования.

Общее методическое замечание 17.9

С помощью механизма классов программист может создавать новые типы. Эти новые типы могут быть спроектированы таким образом, что их будет столь же удобно использовать, как и встроенные типы. Другими словами, C++ — это расширяемый язык. Хотя язык C++ и несложно расширять с помощью новых типов, сам базовый язык изменений не допускает.

Новые абстрактные типы данных, созданные в средах разработки C++, могут быть собственностью отдельных программистов, небольших групп или компаний. Абстрактные типы данных могут также помещаться в стандартные библиотеки классов, предназначенные для широкого распространения. Это не обязательно способствует распространению стандартов, хотя *de facto* делает появление стандартов вполне вероятным. В полной мере значение языка C++ будет осознано только тогда, когда станут широко доступными большие стандартизованные библиотеки классов. Должен быть инициирован формальный процесс, способствующий разработке стандартизованных библиотек. В Соединенных Штатах такая стандартизация часто происходит благодаря ANSI, Американскому национальному институту стандартов. В настоящее время ANSI разрабатывается стандартная версия языка C++. Независимо от того, каким образом, в конечном счете, появятся эти библиотеки, читатель, изучающий C++ и объектно-ориентированное программирование, будет готов воспользоваться преимуществами новых видов ускоренной, ориентированной на компоненты разработки программного обеспечения, которую делают возможной библиотеки абстрактных типов данных.

17.8.2. Пример: абстрактный тип данных — строка

C++ намеренно не является всеобъемлющим языком и предоставляет программистам только исходный материал для создания широкого диапазона систем. Проектировщики языка не хотели вводить в язык ничего, что ограничивало бы его эффективность. Они стремились к тому, чтобы C++ подходил для системного программирования, которое требует эффективного выполнения программ. Конечно, можно было бы включить строковый тип во встроенные типы языка C++, однако проектировщики предпочли вместо этого снабдить язык механизмом для создания и реализации таких абстрактных типов данных посредством классов. Мы разработаем свой собственный абстрактный тип данных для строк в главе 18.

17.8.3. Пример: абстрактный тип данных — очередь

Каждый из нас время от времени стоит в очередях. *Очередью* мы называем ожидание в ряд, один за другим. Мы стоим в очереди в кассу в супермаркете, мы стоим в очереди на бензоколонке, мы стоим в очереди, чтобы сесть в автобус, мы стоим в очереди, чтобы оплатить пошлинный сбор на скоростной магистрали, а все студенты слишком хорошо знают об очередях при регистрации курсов, которые они хотели бы прослушать. Компьютерные системы внутри себя используют много разновидностей очередей, поэтому нам необходимо уметь писать программы, которые моделируют свойства очередей и операции над ними.

Очередь является замечательным примером абстрактного типа данных. Очередь предоставляет своим клиентам хорошо понятную схему поведения. Клиенты по одному помещают объекты в очередь — используя для этого операцию *постановки в очередь* — и по одному получают эти объекты обратно по мере необходимости, используя для этого операцию *исключения из очереди*. Чисто умозрительно очередь может стать бесконечно длинной. Реальная очередь, разумеется, имеет конечную длину. Элементы возвращаются из очереди в порядке *FIFO* («первым пришел, первым вышел»).

Очередь скрывает внутреннее представление данных, в котором каким-то образом предусмотрено отслеживание элементов, ожидающих в очереди в настоящее время, и предлагает своим клиентам набор операций, а именно операцию *постановки в очередь* и операцию *исключения из очереди*. Клиентов не интересует реализация очереди. Клиенты просто хотят, чтобы очередь функционировала «как обещалось». Когда клиент ставит в очередь новый элемент, очередь должна принять этот элемент и как-то поместить его в некоторую структуру данных FIFO. Когда клиенту нужен следующий элемент из начала очереди, очередь должна удалить этот элемент из своего внутреннего представления и передать его внешнему миру в порядке FIFO, т.е. при выполнении оператора исключения из очереди следующим должен быть возвращен элемент, находившийся в очереди наибольшее время.

Абстрактный тип данных очереди гарантирует целостность своей внутренней структуры данных. Клиенты не могут непосредственно манипулировать этой структурой. Только сам абстрактный тип очереди имеет доступ к своим внутренним данным. Клиенты могут инициировать выполнение только допустимых операций над представлением данных: операции, не предусмотренные открытым интерфейсом абстрактного типа данных, отвергаются некоторым адекватным образом. Это может означать выдачу сообщения об ошибке, завершение выполнения программы или просто игнорирование запроса этой операции.

17.9. Контейнерные классы и итераторы

К наиболее популярным типам классов принадлежат *контейнерные классы* (также называемые *классами-коллекциями*), т.е. классы, разработанные для хранения совокупности объектов. Контейнерные классы обычно предоставляют услуги типа вставки, удаления, поиска, сортировки, проверки элемента на принадлежность классу и т.п. Массивы и связанные списки являются примерами контейнерных классов.

Обычно с классом-коллекцией ассоциируют *объекты-итераторы*, или просто *итераторы*. Итератор — это объект, который возвращает следующий элемент коллекции. После реализации итератора класса несложно написать выражение для получения следующего элемента контейнера. Итераторы пишут в виде друзей классов, для которых они выполняют итерации. Подобно тому, как книга, которую читают одновременно несколько читателей, может одновременно содержать несколько закладок, на контейнерном классе может действовать сразу несколько итераторов.

17.10. Шаблоны классов

В главе 15 мы обсуждали шаблоны функций и то, как они способствуют повторному использованию программного обеспечения. Мы завершаем эту главу обсуждением *шаблонов классов*.

Можно понять, что такое стек, независимо от типа элементов, которые в него помещаются. Однако, когда дело доходит до реального программирования стека, должен быть задан тип данных. Это создает чудесную возможность для повторного использования программного обеспечения. Все, что нам нужно — это способ описания понятия стека в обобщенном виде и создания экземпляров классов, которые являются копиями этого обобщенного класса, однако при этом зависят от типа. Такую возможность в C++ предоставляют шаблоны классов.

Общее методическое замечание 17.10

Шаблоны классов способствуют повторному использованию программного обеспечения.

Шаблоны классов были введены в версии 3 компилятора C++ AT&T, так что они являются относительно недавним добавлением к языку.

Шаблоны классов часто называют *параметризованными типами*, поскольку для них требуется один или более параметров-типов, специфицирующих способ настройки обобщенного шаблона.

Программист, желающий использовать шаблоны классов, просто пишет одно обобщенное определение шаблона класса. Всякий раз, когда программисту требуется новый класс, он использует сжатую, несложную нотацию, и компилятор генерирует исходный код для указанного программистом класса. Например, один шаблон классов стека может стать, таким образом, базой для создания многих классов (например, «стека значений типа **float**», «стека значений типа **int**», «стека значений типа **char**» и т.д.), необходимых программе.

Обратите внимание на определение шаблона класса стека на рис. 17.10. Оно выглядит похожим на стандартное определение класса за исключением того, что ему предшествует заголовок

```
template <class T>
```

чтобы показать, что это определение шаблона класса, которое получает один параметр — тип **T**, указывающий тип класса стека, который хочет создать программист. Тип элементов, которые будут храниться в этом стеке, упоминается лишь в обобщенном виде (как **T**) повсюду в заголовке класса стека и в определениях элементов-функций.

```
// TSTACK1.H
// Простой шаблон класса Stack

#ifndef TSTACK1_H
#define TSTACK1_H

template <class T>
class Stack {
public:
   Stack(int = 10);      // Конструктор с размером стека
                         // по умолчанию 10
   ~Stack() { delete [] stackPtr; }  // Деструктор
   int push(const T&);   // Помещает элемент в стек
   int pop(T&);          // Выталкивает элемент из стека
   int isEmpty() const { return top == -1; }  // 1, если пуст
   int isFull() const { return top == size - 1; }
                                                // 1, если полон
private:
   int size;      // количество элементов в стеке
   int top;       // местоположение вершины стека
   T *stackPtr;   // указатель на стек
};

// Конструктор с размером стека по умолчанию 10
template <class T>
Stack<T>::Stack(int s)
{
   size = s;
   top = -1;  // Пустой стек
   stackPtr = new T[size];
}

// Помещает элемент в стек
// возвращает 1 в случае успеха, 0 в противном случае
template <class T>
int Stack<T>::push (const T &item)
{
   if (!isFull()) {
      stackPtr[++top] = item;
      return 1;  // помещение в стек успешно
   }
   return 0;  // поместить в стек не удалось
}
```

Рис. 17.10. Определение шаблона класса **Stack** (часть 1 из 3)

```
// Выталкивает элемент из стека
template <class T>
int Stack<T>::pop(T &popValue)
{
   if (!isEmpty())  {
      popValue = stackPtr[top--];
      return 1;  // выталкивание из стека успешно
   }
   return 0;  // вытолкнуть из стека не удалось
}

#endif

// FIG17_10.CPP
// Программа-тестер для шаблона Stack

#include <iostream.h>
#include "tstack1.h"

main()
{
   Stack<float> floatStack(5);
   float f = 1.1;
   cout << "Pushing elements onto floatStack\n";

   while (floatStack.push(f))  { // успех (возвращена 1)
      cout << f << ' ';
      f += 1.1;
   }

   cout << "\nStack is full. Cannot push " << f
        << "\n\nPopping elements from floatStack" << endl;

   while (floatStack.pop(f))  // успех (возвращена 1)
      cout << f << ' ';

   cout << "\nStack is empty. Cannot pop" << endl;

   Stack<int> intStack;
   int i = 1;
   cout << "\nPushing elements onto intStack\n";

   while (intStack.push(i))  { // успех (возвращена 1)
      cout << i << ' ';
      i++;
   }

   cout << "\nStack is full. Cannot push " << i
        << "\n\nPopping elements from intStack" << endl;

   while (intStack.pop(i))  // успех (возвращена 1)
      cout << i << ' ';

   cout << "\nStack is empty. Cannot pop" << endl;
   return 0;
}
```

Рис. 17.10. Определение шаблона класса **Stack** (часть 2 из 3)

```
Pushing elements onto floatStack
1.1 2.2 3.3 4.4 5.5
Stack is full. Cannot push 6.6

Popping elements from floatStack
5.5 4.4 3.3 2.2 1.1
Stack is empty. Cannot pop

Pushing elements onto intStack
1 2 3 4 5 6 7 8 9 10
Stack is full. Cannot push 11

Popping elements from intStack
10 9 8 7 6 5 4 3 2 1
Stack is empty. Cannot pop
```

Рис. 17.10. Определение шаблона класса **Stack** (часть 3 из 3)

Теперь давайте рассмотрим тестовую программу (функцию **main**) для шаблона классов стека. Программа начинается с создания экземпляра объекта **floatStack** размера 5. Этот объект объявляется объектом класса **Stack<float>**. Компилятор ассоциирует тип **float** с параметризованным типом **T** шаблона и генерирует исходный код для класса стека типа **float**.

Затем в программе значения типа **float** 1.1, 2.2, 3.3, 4.4 и 5.5 последовательно помещаются в **floatStack**. Цикл помещения в стек завершается при попытке программы поместить шестое значение в стек **floatStack** (который уже полон, поскольку он был создан с 5-ю элементами).

Далее в программе эти пять значений выталкиваются из стека (в порядке LIFO). Программа пытается вытолкнуть и шестое значение, однако **floatStack** уже пуст и, таким образом, цикл выталкивания завершается.

После этого создается экземпляр целочисленного стека с помощью объявления

```
Stack<int> intStack;
```

Поскольку не было задано никакого размера, размер принимает значение по умолчанию 10, как было определено в конструкторе по умолчанию. Опять программа в цикле помещает значения в **intStack** до тех пор, пока стек не будет заполнен, и затем в цикле выталкивает значения из **intStack**, пока стек не станет пустым.

Все определения элементов-функций *вне заголовка шаблона класса* начинаются с

```
template <class T>
```

В дальнейшем каждое определение похоже на стандартное определение за исключением того, что тип элемента стека всегда указывается обобщенно в виде типа-параметра **T**. Операция разрешения области действия, как всегда, используется для привязки определения каждой функции-элемента к области действия соответствующего класса. В данном случае именем класса является **Stack<T>**. Когда создается экземпляр объекта **floatStack** типа **Stack<float>**, конструктор **Stack** создает для представления стека массив элементов типа **T**. Вместо **T** подставляется тип **float** и из оператора

```
stackPtr = new T[size];
```

компилятор генерирует в версии **Stack<float>** оператор вида

```
stackPtr = new float [size];
```

Параметризованные типы (т.е. шаблоны) могут принимать более одного параметра, например:

```
template <class T, class S, class R>
```

Каждому параметру должно предшествовать ключевое слово **class**.

Резюме

- Ключевое слово **const** может быть использовано для указания на то, что объект не является модифицируемым и что любая попытка его изменения является ошибкой.
- Компилятор C++ запрещает вызовы не-константных функций для константных объектов.
- Константная функция-элемент не может изменять объект.
- Функция специфицируется в качестве константной как в ее объявлении, так и в определении.
- Константная функция-элемент может быть перегружена своей не-константной версией. Компилятор автоматически выбирает, какая из перегруженных функций должна быть использована, исходя из того, был ли объект объявлен как **const**.
- Константный объект должен быть инициализирован.
- Если класс содержит константные элементы-объекты, то конструктору должны быть предоставлены инициализаторы элементов.
- Классы могут содержать объекты других классов.
- Элементы данных — объекты создаются до создания объектов включающего их класса.
- Если для объекта — элемента данных не задан инициализатор, то для этого объекта вызывается конструктор по умолчанию.
- Дружественная функция класса определяется вне этого класса и имеет право доступа к его закрытым и защищенным элементам.
- Объявления дружественных функций и классов могут быть помещены в любое место определения класса.
- Каждая перегруженная функция, которую собираются сделать дружественной, должна быть явно объявлена в качестве друга класса.
- Каждый объект поддерживает указатель на самого себя — называемый указателем **this** — который становится неявным параметром во всех ссылках на элементы внутри этого объекта. Указатель **this** неявно используется для ссылки как на элементы-функции, так и элементы данных объекта.
- Каждый объект может определить свой собственный адрес, используя ключевое слово **this**.
- Указатель **this** может быть использован явно, однако чаще всего он используется неявно.

- Операция **new** автоматически создает объект надлежащего размера и возвращает указатель соответствующего типа. Для освобождения памяти, выделенной этому объекту, в C++ используется операция **delete**.
- Статический элемент данных хранит «общеклассовую» информацию. Объявление статического элемента начинается с ключевого слова **static**.
- Статические элементы данных имеют область действия класса.
- К статическим элементам класса можно обращаться через объект этого класса, либо посредством имени класса с операцией разрешения области действия.
- Функция-элемент может быть объявлена статической, если она не обращается к не-статическим элементам класса. В отличие от не-статических, статическая функция-элемент не имеет указателя **this**. Это происходит потому, что статические элементы данных и статические элементы-функции существуют независимо от объектов класса.
- Шаблоны классов предоставляют программисту способ описания понятия класса в обобщенном виде и создания экземпляров классов, которые являются копиями этого обобщенного класса, однако при этом зависят от типа.
- Шаблоны классов часто называют параметризованными типами, поскольку для них требуется один или более параметров-типов, специфицирующих способ настройки обобщенного шаблона.
- Определения шаблонов классов начинаются с заголовка

```
template <class T>
```

на строке, предшествующей определению класса. Может объявляться более одного параметра-типа. В этом случае они разделяются запятыми и каждому типу предшествует ключевое слово **class**.

- Экземпляры шаблона класса создаются путем указания типа класса, которое следует за именем шаблона, как в следующем примере:

```
ClassName<typeName> objectName;
```

Компилятор подставляет **typeName** вместо типа-параметра по всему определению класса и определениям элементов-функций.

Терминология

вложенный класс
двухместная операция разрешения области действия (::)
деструктор
динамические объекты
дружественная функция
дружественный класс
инициализатор элемента
итератор
композиция
конкатенация вызовов элементов-функций
константная функция-элемент
константный объект
конструктор
конструктор объекта-элемента
конструктор по умолчанию
контейнерные классы
область действия класса
объект – элемент данных
операция **delete**
операция **delete[]**
операция **new**
операция выбора элемента — стрелка
операция выбора элемента — точка (.)

параметризованный тип
принцип минимума привилегий
расширяемость
спецификаторы доступа к элементам
статическая функция-элемент
статический элемент данных
указатель **this**
указатель на элемент
шаблон класса

Распространенные ошибки программирования

17.1. Определение с модификатором **const** элемента-функции, которая изменяет элемент данных объекта.

17.2. Определение с модификатором **const** элемента-функции, которая вызывает не-константную функцию-элемент.

17.3. Вызов не-константной функции-элемента для константного объекта.

17.4. Попытка изменения константного объекта.

17.5. Не предусматривают инициализатор элемента для константного объекта — элемента данных.

17.6. Не объявлен конструктор по умолчанию для объекта — элемента данных, когда для него нет инициализатора элемента. Компилятор в этом случае генерирует ошибку.

17.7. Попытка применить операцию выбора элемента (.) к указателю на объект (операция выбора элемента может быть использована только с объектом или ссылкой на объект).

17.8. Смешивание динамического распределения памяти в стиле **new** и **delete** с вызовами **malloc** и **free**. Память, выделенная **malloc**, не может быть освобождена с помощью **delete**; объекты, созданные с помощью **new**, не могут быть удалены функцией **free**.

17.9. Обращение к указателю **this** внутри статической функции-элемента.

17.10. Объявление статической функции-элемента с модификатором **const**.

Хороший стиль программирования

17.1. Объявляйте константными все элементы-функции, предназначенные для использования с константными объектами.

17.2. Помещайте все объявления дружественных функций и классов в начале области определения класса сразу после его заголовка и не помещайте перед этими объявлениями никаких спецификаторов доступа.

17.3. Хотя в программах на C++ может применяться как память, выделяемая с помощью **malloc** и освобождаемая с помощью **free**, так и объекты, создаваемые с помощью **new** и удаляемые с помощью **delete**, все же лучше использовать только **new** и **delete**.

Советы по повышению эффективности

17.1. Явно инициализируйте объекты — элементы данных посредством инициализаторов элементов. Это устраняет накладные расходы, связанные с двойной инициализацией элементов-объектов, — первый раз при вызове конструктора по умолчанию для объекта — элемента данных и снова при инициализации элемента с помощью set-функций.

17.2. Из-за соображений экономии памяти для класса существует только одна копия каждого элемента-функции, которая вызывается всеми объектами этого класса. С другой стороны, каждый объект имеет собственную копию элементов данных класса.

17.3. Используйте статические элементы данных с целью экономии памяти, если достаточно только одной копии данных.

Общие методические замечания

17.1. Объявление объекта с модификатором const способствует последовательному проведению в жизнь принципа минимума привилегий. Случайные попытки изменить объект выявляются во времена компиляции и не приводят к ошибкам времени выполнения.

17.2. Константная функция-элемент может быть перегружена своей не-константной версией. Компилятор автоматически выбирает, какая из перегруженных функций должна быть вызвана, исходя из того, был ли объект объявлен как **const**.

17.3. Как константные объекты, так и константные «переменные» необходимо инициализировать, используя синтаксис инициализатора элемента. Присваивания в этом случае не допускаются.

17.4. Одной из форм повторного использования программного кода является композиция, когда класс содержит в качестве элементов объекты других классов.

17.5. Если объект содержит несколько объектов в качестве элементов данных, эти элементы данных — объекты создаются в том порядке, в котором они были определены. Тем не менее, не пишите кода, который зависит от специфического порядка выполнения конструкторов этих объектов.

17.6. Понятия типа доступа — **private**, **protected** и **public** — не имеют смысла для объявлений дружественных функций и классов, поэтому эти объявления могут размещаться в любом месте определения класса.

17.7. Некоторые программисты считают, что «дружественность» разрушает сокрытие информации и ослабляет значимость объектно-ориентированного подхода к проекту.

17.8. Статические элементы данных и статические элементы-функции существуют и к ним можно обращаться, даже если не было создано ни одного представителя этого класса.

17.9. С помощью механизма классов программист может создавать новые типы. Эти новые типы могут быть спроектированы таким образом, что их будет столь же удобно использовать, как и встроенные типы. Таким образом, C++ — это расширяемый язык. Хотя язык C++ и несложно расширять с помощью новых типов, сам базовый язык изменений не допускает.

17.10. Шаблоны классов способствуют повторному использованию программного обеспечения.

Упражнения для самоконтроля

17.1. Заполните пропуски в каждом из следующих утверждений:

a) Синтаксис _____ используется для инициализации константных элементов класса.

b) _____ должен быть объявлен в классе, чтобы иметь доступ к закрытым элементам данных класса.

c) Операция _____ возвращает _____ на динамически созданный объект класса.

d) Константный объект должен быть _____; после создания он не может быть изменен.

e) _____ элемент данных хранит «общеклассовую» информацию.

f) Каждый объект поддерживает указатель на себя самого, называемый указателем _____.

g) _____ обеспечивают средства описания понятия класса в обобщенном виде.

h) Ключевое слово _____ указывает, что объект или переменная являются немодифицируемыми.

i) Если не предусмотрен инициализатор для элемента — объекта класса, то вызывается _____ объекта.

j) Функция-элемент может быть статической, если она не обращается к _____ элементам класса.

k) Дружественные функции могут обращаться к _____ и _____ элементам класса.

l) Шаблоны классов иногда называют _____ типами.

m) Элементы данных — объекты класса создаются _____ объекта включающего их класса.

n) Операция _____ возвращает память, предварительно выделенную операцией **new**.

17.2. Найдите ошибку(и) в каждом из следующих случаев и объясните, как ее можно исправить.

a)
```
char *string;
string = new char[20];
free(string);
```

b)
```
class Example {
public:
```

```
   Example(int y = 10)  { data = y; }
   int getIncrementedData() const { return ++data; }
   static int getCount()
   {
      cout << "Data is " << data << endl;
      return count;
   }
private:
   int data;
   static int count;
};
```

Ответы к упражнениям для самоконтроля

17.1. a) инициализатора элемента. b) **friend**. c) **new**, указатель. d) инициализирован. e) Статический. f) **this**. g) Шаблоны классов. h) **const**. i) конструктор по умолчанию. j) не-статическим. k) закрытым, защищенным. l) параметризованными. m) до. n) **delete**.

17.2. a) Ошибка: память, динамически выделенная операцией **new**, освобождается с помощью стандартной библиотечной функции языка C **free**.

Исправление: используйте для возвращения памяти операцию **delete** языка C++. Динамическое распределение памяти в стиле языка C не следует смешивать с операциями C++ **new** и **delete**.

b) Ошибка: в определении класса **Example** содержится две ошибки. Первая появляется в функции **getIncrementedData**. Эта функция объявлена с модификатором **const**, однако она изменяет объект.

Исправление: чтобы исправить первую ошибку, уберите ключевое слово **const** из определения функции **getIncrementedData**.

Ошибка: вторая ошибка находится в функции **getCount**. Эта функция объявлена статической, поэтому ей не разрешен доступ к не-статическим элементам класса.

Исправление: уберите строку, в которой выводятся данные, из определения функции **getCount**.

Упражнения

17.3. Сравните динамическое распределение памяти с помощью операций C++ **new** и **delete** и распределение памяти с использованием стандартных библиотечных функций языка C **malloc** и **free**.

17.4. Объясните понятие дружественности в языке C++. Объясните отрицательные аспекты дружественности, как это описано в книге.

17.5. Может ли корректное определение класса **Time** включать оба нижеследующих конструктора?

```
Time(int h = 0, int m = 0, int s = 0);
Time();
```

17.6. Что происходит, если для конструктора или деструктора определяется тип возвращаемого значения (даже **void**)?

17.7. Создайте класс **Date** со следующими возможностями:

a) Выведите дату в нескольких форматах, например:

```
DDD YYYY
MM/DD/YY
June 14, 1992
```

b) Используйте перегруженные конструкторы для создания объектов **Date**, инициализированных датами в форматах из пункта (a).

c) Создайте конструктор класса **Date**, который считывает системную дату, используя стандартные библиотечные функции заголовочного файла **time.h**, и устанавливает элементы **Date**.

В главе 18 мы сможем создавать операции для проверки двух дат на равенство и для сравнения дат, чтобы определить, предшествует ли одна дата другой или следует за ней.

17.8. Руководствуясь программой обработки списка на рис. 12.3, постройте класс **List**. Этот класс должен обеспечивать те же возможности, что и в примере. Напишите программу-тестер для тщательной проверки класса.

17.9. Руководствуясь программой обработки очереди на рис. 12.13, постройте класс **Queue**. Этот класс должен обеспечивать те же возможности, что и в примере. Напишите программу-тестер для тщательной проверки класса.

17.10. Используя в качестве отправной точки класс **List**, разработанный в упражнении 17.8, создайте шаблон класса **List**, способный создавать экземпляры объектов класса **List** для хранения различных типов данных. Напишите программу-тестер для тщательной проверки шаблона.

17.11. Используя в качестве отправной точки класс **Queue**, разработанный в упражнении 17.9, создайте шаблон класса **Queue**, способный создавать экземпляры объектов класса **Queue** для хранения различных типов данных. Напишите программу-тестер для тщательной проверки шаблона.

глава

18

Перегрузка операций



Цели

- Понять принципы переопределения операций для работы с новыми классами.
- Понять, как объекты одного класса преобразуются в другой класс.
- Изучить случаи, когда перегрузка операций целесообразна, и когда она нецелесообразна.
- Изучить несколько интересных классов, использующих перегруженные операции.

Содержание

18.1. Введение
18.2. Основные принципы перегрузки операций
18.3. Запреты на перегрузку операций
18.4. Функции-операции как элементы класса и как дружественные функции
18.5. Перегрузка операций передачи в поток и извлечения из потока
18.6. Перегрузка одноместных операций
18.7. Перегрузка двухместных операций
18.8. Пример: класс Array
18.9. Преобразование типов
18.10. Пример: класс String
18.11. Перегрузка ++ и --
18.12. Пример: класс Date

Резюме • Распространенные ошибки программирования • Хороший стиль программирования • Советы по повышению эффективности • Общие методические замечания • Упражнения для самоконтроля • Ответы к упражнениям для самоконтроля • Упражнения

18.1. Введение

В 16-й и 17-й главах мы познакомились с классами C++ и с абстрактными типами данных (ADT). Операции с объектами класса (т.е. представителями ADT) осуществлялись посредством посылки сообщений объектам (в форме вызова функций-элементов). Нотация такого вызова функции становится громоздкой для определенных категорий классов, в особенности для математических. Чтобы оперировать с объектами таких классов, было бы удобно использовать широкий набор встроенных операций C++. В этом разделе мы покажем, как разрешить операциям C++ работать с объектами класса. Этот процесс называется *перегрузкой операций* и является прямым и естественным путем к расширению C++ за счет этих новых возможностей.

Операция << в C++ может иметь различный смысл и используется в качестве операции передачи в поток и операции сдвига влево. Это пример *перегруженной операции*. Операция >> также перегружена, поскольку она используется как операция извлечения из потока и как операция правого сдвига. Обе эти операции перегружаются в библиотеке классов C++. Сам по себе C++ уже перегружает операции + и -. Эти операции выполняют различные функции, которые зависят от контекста применения — в целочисленной арифметике, арифметике чисел с плавающей точкой и арифметике указателей.

В целом C++ дает программисту возможность перегружать большинство операций, так чтобы они были чувствительны к контексту, в котором применяются. Компилятор генерирует соответствующий программный код, исходя из способа применения данной операции. Некоторые операции перегружаются достаточно часто, особенно операции присваивания и различные арифметические операции, такие как + и -. Действие, выполняемое перегруженной операцией, может также быть выполнено посредством явного обращения к функции, но, как правило, нотация операции легче читается.

Мы обсудим случаи, когда следует использовать перегрузку операции и когда этого делать не следует. Ниже будет приведено достаточное количество законченных программ, которые иллюстрируют технику перегрузки операций.

18.2. Основные принципы перегрузки операций

Программирование в C++ является процессом, чувствительным к типу данных и ориентированным на типы данных. Программист может использовать встроенные типы и может определять новые. Встроенные типы могут использоваться с широким набором операций C++. Операции дают программистам сжатую нотацию манипуляций с объектами встроенных типов.

Программист также может использовать операции с типами, определенными пользователем. Несмотря на то, что C++ не позволяет создавать новые операции, он позволяет перегружать существующие операции и, когда они применяются к объектам класса, операции приобретают смысл, соответствующий новым типам. Это очень сильная сторона C++. Перегрузка операций способствует расширяемости C++ и, несомненно, является одной из наиболее привлекательных особенностей этого языка программирования.

Хороший стиль программирования 18.1

Используйте перегрузку операции, когда это помогает сделать программу яснее, чем при выполнении тех же действий посредством явного вызова функции.

Хороший стиль программирования 18.2

Избегайте избыточного или непоследовательного использования перегрузки операций, так как это может сделать программу трудно читаемой.

Несмотря на то, что перегрузка операций может восприниматься как некая экзотическая возможность, большинство программистов неявно используют перегруженные операции. Например, операция плюс (+) работает существенно различным образом с целыми, числами с плавающей точкой и числами

с двойной точностью. Но плюс, тем не менее, прекрасно работает с переменными типа **int**, **float** и **double**, а также с числами других встроенных типов, поскольку операция плюс (+) перегружена уже в самом языке C++.

Операции перегружаются посредством написания обычного определения функции (с заголовком и телом) за исключением того, что именем функции становится ключевое слово **operator** с последующим символом перегружаемой операции. Например, имя функции **operator+** могло бы использоваться для перегрузки аддитивной операции.

Чтобы использовать операцию с объектами класса, эта операция *должна* быть перегружена, но здесь есть два исключения. Операция присваивания (=) может использоваться с любым классом без перегрузки. Поведение по умолчанию операции присваивания (в тех случаях, когда она не перегружена) обеспечивается *поэлементным копированием* элементов данных класса. Далее мы увидим, что такая поэлементная копия, создаваемая по умолчанию, представляет опасность для классов, элементы которых ссылаются на динамически выделяемые области памяти; для таких классов мы, как правило, будем перегружать операцию присваивания. Операция адреса (&) также может использоваться с объектами любого класса без перегрузки; эта операция просто возвращает адрес объекта в памяти. Операция адреса может быть также перегружена.

Перегрузка в наибольшей степени подходит для математических классов. Они часто требуют перегрузки большого перечня операций, чтобы сохранялась согласованность со способом обработки математических объектов, которые существуют в реальной жизни. Например, было бы странно перегружать только операцию суммирования для класса комплексных чисел, поскольку другие арифметические операции над комплексными числами производятся не менее часто.

Язык C++ содержит богатый набор операций. Программисты, работающие с C++, понимают смысл и контекст каждой операции. Так что, если дело доходит до перегрузки операций для нового класса, программист скорее всего примет достаточно оправданное решение.

Ключевой пункт при выполнении перегрузки операции — обеспечение той же лаконичности выражений для типов, определяемых пользователем, которую обеспечивает C++ для встроенных типов за счет богатого набора операций. Перегрузка операций не является автоматическим процессом, и для выполнения нужной процедуры программист должен написать перегружающую операцию функцию. Иногда эту роль лучше выполняют функции-элементы, иногда дружественные функции и, время от времени, эту роль могут выполнять функции, не являющиеся ни элементами, ни друзьями.

Неправильным случаем перегрузки была бы перегрузка операции + с целью выполнения операции, аналогичной вычитанию, или перегрузка / для выполнения операции, аналогичной умножению. Такого рода перегрузка может существенно затруднить понимание программы.

Хороший стиль программирования 18.3

Перегружайте операцию для выполнения тех же действий с объектами класса, или близких по смыслу действий, которые выполняет эта операция со встроенными типами.

Хороший стиль программирования 18.4

Перед тем как писать программу на C++, перегружающую операции, изучите руководство по вашему компилятору, чтобы учесть различные ограничения и требования, накладываемые на конкретные операции.

18.3. Запреты на перегрузку операций

Большая часть операций C++ может быть перегружена. Эти операции приведены на рис. 18.1. На рис. 18.2 приведены операции, которые не могут перегружаться.

Операции, которые могут быть перегружены							
+	-	*	/	%	^	&	|
~	!	=	<	>	+=	-=	*=
/=	%=	^=	&=	|=	<<	>>	>>=
<<=	==	!=	<=	>=	&&	||	++
--	->*	,	->	[]	()	new	delete

Рис. 18.1. Операции, которые могут быть перегружены

Операции, которые не могут быть перегружены				
.	.*	::	?:	sizeof

Рис. 18.2. Операции, которые не могут быть перегружены

Приоритет операций не может быть изменен посредством перегрузки. Это может привести к неудобным ситуациям, когда операция перегружается так, что ее фиксированный приоритет плохо соответствует смыслу выполняемых действий. Тем не менее, остается возможность использовать скобки для определения порядка оценки перегруженных операций в выражении.

Ассоциативность операций также не может быть изменена посредством перегрузки. Для перегрузки операций нельзя использовать аргументы по умолчанию.

Отсутствует возможность изменить число операндов, которое подразумевает операция. Одноместная операция остается одноместной и при перегрузке, а двухместная остается двухместной. Единственная трехместная операция C++, условная (?:), не может быть перегружена. Каждая из операций &, *, + и - имеет одноместную и двухместную формы, которые могут перегружаться раздельно.

Нельзя создавать новые операции; только существующие операции могут быть перегружены. Это запрещает программисту пользоваться такими популярными нотациями, как операция **, означающая в языке Basic возведение в степень.

Распространенная ошибка программирования 18.1

Попытка создать новую операцию.

Способ работы операции с объектами встроенных типов не может быть изменен посредством перегрузки. Например, программист не может изменить способ суммирования операцией + двух целых. Перегрузка операций работает только с объектами, тип которых определен пользователем, или в смешанных ситуациях, когда объект пользовательского типа участвует в операции вместе с объектом встроенного типа.

Распространенная ошибка программирования 18.2

Попытка модифицировать способ работы операции с объектами встроенных типов.

Общее методическое замечание 18.1

По меньшей мере один аргумент функции-операции должен быть объектом класса или должен ссылаться на объект класса. Это делает невозможным изменение способа воздействия операции на объекты встроенных типов.

При перегрузке ( ), [ ], -> или = перегружающая операцию функция должна объявляться как функция-элемент. Для других операций перегружающие функции могут быть друзьями.

Перегрузка операции присваивания и операции суммирования с целью разрешить такие операции, как

```
object2 = object2 + object1;
```

не означает, что автоматически будет перегружена операция +=, чтобы выполнялся такой оператор, как

```
object2 += object1;
```

Однако такое поведение может быть достигнуто посредством явной перегрузки операции += для этого класса.

Распространенная ошибка программирования 18.3

Предположение, что перегрузка операций (таких как +) автоматически перегружает операции присваивания (такие как +=). Операции могут быть перегружены только явным образом; неявная перегрузка отсутствует.

18.4. Функции-операции как элементы класса и как дружественные функции

Функции-операции могут быть функциями-элементами или не быть ими; функции — не элементы обычно являются друзьями. Функции-элементы используют неявный указатель **this**, чтобы получить один из аргументов-объектов своего класса. Этот аргумент класса должен быть указан явным образом при вызове функции, не являющейся элементом класса.

Независимо от того, объявляется ли функция-операция как функция-элемент или как функция, не являющаяся элементом, в выражении операция используется одинаковым образом. Так какой же вариант лучше?

Когда функция-операция объявляется в качестве функции-элемента, левый (или единственный) операнд должен быть объектом (или ссылкой на объект), принадлежащим классу этой операции. Если необходимо, чтобы левый

операнд был объектом другого класса или объектом встроенного типа, эта функция-операция должна объявляться как функция, не являющаяся элементом класса, так, как показано в разделе 18.5, где проиллюстрирована перегрузка << и >> в качестве операций передачи и извлечения из потока. Функции-операции, объявляемой в качестве функции, не являющейся элементом класса, нужно быть другом, если она должна получать прямой доступ к закрытым или защищенным элементам класса.

Перегруженная операция << должна иметь в качестве левого операнда тип **ostream &** (такой, как **cout** в выражении **cout << classObject**), так что она должна быть функцией, не являющейся элементом класса. Аналогично операция >> должна иметь в качестве левого операнда тип **istream &** (такой, как **cin** в выражении **cin >> classObject**) и не являться функцией-элементом. Кроме того, каждая из этих перегружающих операцию функций требует доступа к закрытым элементам данных класса объектов, который должен выводиться или вводиться. Так что эти перегружающие операцию функции делаются обычно дружественными функциями класса по причинам, связанным с эффективностью.

Совет по повышению эффективности 18.1

Операцию можно перегрузить как функцию, не являющуюся элементом или дружественной функцией, но, если такой функции требуется доступ к закрытым или защищенным данным класса, то ей придется обращаться к открытому интерфейсу этого класса (функциям вроде get или set). Такое опосредованное обращение к данным может привести к потере производительности.

Функция-операция — элемент класса вызывается только тогда, когда левый операнд двухместной операции является объектом именно этого класса или когда единственный операнд одноместной операции является объектом этого класса.

Другая причина, по которой для перегрузки операции можно выбрать функцию, не являющуюся элементом — требование коммутативности операции. Например, у нас есть объект **number** типа **long int** и объект **bigInteger1**, принадлежащий к классу **HugeInteger** (класс, в котором целые могут быть произвольно большими вне зависимости от ограничений разрядности аппаратной части; класс **HugeInteger** разрабатывается в разделе упражнений). Операция суммирования (+) генерирует временный объект **HugeInteger** как сумму **HugeInteger** и **long int** (согласно выражению **bigInteger1 + number**), или как сумму **long int** и **HugeInteger** (согласно выражению **number + bigInteger1**). Таким образом, мы требуем, чтобы операция суммирования была коммутативна (как и в обычной ситуации). Проблема состоит в том, что объект класса должен стоять в левой части суммы, если операция перегружена как функция-элемент. Таким образом, чтобы разрешить **HugeInteger** находиться в правой части суммы, мы перегружаем операцию как друга. Функция **operator+**, которая обрабатывает **HugeInteger** с левой стороны, может оставаться функцией-элементом.

18.5. Перегрузка операций передачи в поток и извлечения из потока

В C++ имеется возможность ввода и вывода стандартных типов данных с использованием операций извлечения из потока >> и передачи в поток <<. Эти операции являются перегруженными (в библиотеке классов, поставляемых с компиляторами C++), и могут обрабатывать любой стандартный тип данных, включая строки и адреса памяти. Кроме того, операции передачи и извлечения из потока могут быть перегружены, чтобы выполнять ввод и вывод типов, определяемых пользователем. На рис. 18.3 показана перегрузка операций передачи и извлечения из потока, позволяющая им обрабатывать данные класса телефонного номера (определенного пользователем) **PhoneNumber**. В этой программе предполагается, что телефонный номер введен корректно. Проверку корректности ввода номера мы оставляем читателю в качестве упражнения.

Функция-операция извлечения из потока (**operator>>**) принимает в качестве аргументов ссылку (**input**) на **istream** и ссылку (**num**) на тип, определяемый пользователем (**PhoneNumber**), и возвращает ссылку на **istream**. На рис. 8.13 функция-операция **operator>>** используется для ввода телефонного номера в формате

```
(800) 555-1212
```

в объект класса **PhoneNumber**. Когда компилятор встречает в **main** выражение

```
cin >> phone
```

он генерирует вызов функции

```
operator>>(cin, phone);
```

Когда выполняется этот вызов, параметр **input** становится псевдонимом для **cin**, а параметр **num** становится псевдонимом для **phone**. Функция-операция использует функцию-элемент **getline** класса **istream** для чтения трех частей телефонного номера в качестве строк в элементы **areaCode**, **exchange** и **line** объекта типа **PhoneNumber** (ссылки **num** в функции-операции и **phone** в **main**). Функция **getline** обсуждается детально в главе 21. Символы скобок, пробела и тире пропускаются вызовом **ignore** (функции-элемента **istream**), которая отбрасывает заданное число символов во входном потоке (один символ по умолчанию). Функция **operator>>** возвращает **input** (т.е. **cin**) как ссылку на **istream**. Это разрешает *конкатенацию* операций ввода объектов **PhoneNumber** с операциями ввода других объектов **PhoneNumber** или объектов других типов. Например, два объекта **PhoneNumber** могут вводиться следующим образом:

```
cin >> phone1 >> phone2
```

Сначала выполнилось бы выражение **cin >> phone1** посредством вызова функции

```
operator>>(cin, phone1);
```

Этот вызов вернул бы затем **cin** как значение выражения **cin>>phone1**, так что оставшаяся часть выражения интерпретировалась бы просто как **cin>>phone2**.

```
// FIG18_3.CPP
// Перегрузка операций передачи в поток
// и извлечения из потока
#include <iostream.h>

class PhoneNumber {
   friend ostream &operator<<(ostream &, const PhoneNumber &);
   friend istream &operator>>(istream &, PhoneNumber &);
private:
   char areaCode[4];// 3-цифры и null
   char exchange[4];// 3-цифры и null
   char line[5];// 4-цифры и null
};

// Перегрузка операции передачи в поток (не может быть
// функцией-элементом).
ostream &operator<<(ostream &output, const PhoneNumber &num)
{
   output << "(" << num.areaCode << ") "
          << num.exchange << "-" << num.line;

   return output;      // разрешить cout << a << b << c;
}

// перегрузка операции извлечения из потока
istream &operator>>(istream &input, PhoneNumber &num)
{
   input.ignore();                    //пропуск (
   input.getline(num.areaCode, 4); //ввести area code
   input.ignore(2);                   //пропуск ) и пробела
   input.getline(num.exchange, 4); //ввод exchange
   input.ignore() ;                   //пропустить тире (-)
   input.getline(num.line, 5);        //ввести line

   return input;       // разрешить cin >> a >> b >> c;
)

main()
{
   PhoneNumber phone; // создать объект phone

   cout << "Enter a phone number in the "
     << "form (123) 456-7890:\n";

   // cin >> phone вызывает функцию operator>>
   // вызывая operator>>(cin, phone).
   cin >> phone;

   // cout << phone вызывает функцию operator<<
   // вызывая operator<<(cout, phone).
   cout << "The phone number entered was:\n"
        << phone << endl;
   return 0;
}
```

Рис. 18.3. Определяемые пользователем операции передачи и извлечения из потока (часть 1 из 2)

```
Enter a phone number in the form (123) 456-7890:
(800) 555-1212
The phone number entered was:
(800) 555-1212
```

Рис. 18.3. Определяемые пользователем операции передачи и извлечения из потока (часть 2 из 2)

Операция передачи в поток принимает в качестве аргументов ссылку (**output**) на **osrteam** и ссылку (**num**) на тип, определяемый пользователем (**PhoneNumber**), и возвращает ссылку на **ostream**. Функция **operator<<** выводит объекты типа **PhoneNumber**. Когда компилятор встречает в **main** выражение

```
cout << phone
```

он генерирует вызов функции

```
operator<<(cout, phone);
```

Функция **operator<<** выводит части телефонного номера как строки, т.к. они хранятся в формате строки (функция-элемент **getline** из **istream** записывает нулевой символ после завершения ввода).

Обратите внимание, что функции **operator>>** и **operator<<** объявляются в классе **PhoneNumber** в качестве дружественных функций, не являющихся элементами. Эти операции не могут быть элементами класса, т.к. объекты класса **PhoneNumber** всегда появляются в операции в качестве правого операнда; операнд класса должен стоять в левой части, чтобы можно было перегружать операцию как функцию-элемент. Перегруженные операции ввода и вывода должны объявляться как друзья, если им необходимо иметь прямой доступ к элементам класса, не объявленным как **public**.

Общее методическое замечание 18.2

Новые возможности ввода/вывода типов, определяемых пользователем, могут быть введены в C++ без модификации объявлений или элементов данных **private** в любом из классов **ostream** или **istream**. Это способствует расширяемости C++ как языка программирования, что является одним из наиболее привлекательных аспектов C++.

18.6. Перегрузка одноместных операций

Одноместная операция для класса может быть перегружена как не-статическая функция-элемент, не имеющая аргументов, или как функция, не являющаяся элементом и имеющая один аргумент. Этот аргумент должен быть либо объектом класса, либо ссылкой на него.

Позже в этой главе мы будем перегружать одноместную операцию !, чтобы проверить, является ли строка пустой. При перегрузке одноместной операции, например, ! в качестве не имеющей аргументов не-статической функции-элемента компилятор, когда он встречает выражение **!s**, генерирует вызов **s.operator!( )**. Операнд **s** является объектом класса, для которого вызывается функция-элемент **operator!**:

```
class String {
public:
   int operator!() const;
   ...
```

Одноместная операция, такая как **!**, может быть перегружена в качестве функции, не являющейся элементом и имеющей один аргумент, двумя различными способами: либо с аргументом, являющимся объектом, либо с аргументом, который ссылается на объект. Если **s** является объектом класса, то запись **!s** воспринимается точно так же, как вызов **operator!(s)**.

```
class String {
   friend int operator!(const String &);
   ...
};
```

Хороший стиль программирования 18.5

При перегрузке одноместных операций предпочтительнее делать функции-операции элементами класса, а не дружественными функциями, не являющимися элементами. Это более аккуратное решение. Нужно избегать дружественных функций и дружественных классов, если только это не является крайне необходимым. Использование друзей нарушает инкапсуляцию класса.

18.7. Перегрузка двухместных операций

Двухместная операция может быть перегружена как не-статическая функция-элемент с одним аргументом или как функция, не являющаяся элементом и имеющая два аргумента (один из этих аргументов должен быть либо объектом класса, либо ссылкой на него).

Позже в этой главе мы будем перегружать операцию **+=**, чтобы выполнять конкатенацию двух объектов-строк. Если **y** и **z** являются объектами класса, то, когда двухместная операция **+=** перегружается как не-статическая функция-элемент класса **String** с одним аргументом, выражение **y += z** воспринимается точно так же, как **y.operator+=(z)**.

```
class String {
public:
   String &operator+=(const String &);
   ...
};
```

Двухместная операция **+=** может быть перегружена как функция с двумя аргументами, не являющаяся элементом. Один из аргументов должен быть объектом класса или ссылкой на объект. Если **y** и **z** являются объектами класса, тогда **y += z** воспринимается в программе точно так же, как вызов **operator+=(y, z)**.

```
class String {
   friend int &operator+=(String &, const String &);
   ...
};
```

18.8. Пример: класс Array

Нотация массива в C++ является просто альтернативой указателям, поэтому массивы никак не защищены в отношении возможных ошибок. Например, программа может просто «проскочить» массив, поскольку C и C++ не проверяют, вышел индекс за пределы массива или нет. В массиве размера *n* элементы

должны нумероваться следующим образом: 0,..., *n*-1. Альтернативные пределы индексации не разрешены. За один раз нельзя ввести или вывести массив в целом; каждый элемент должен считываться и записываться индивидуально. Два массива нельзя сравнивать, используя операцию равенства или отношения. Когда массив передается функции общего назначения, предназначенной для обработки массивов любого размера, размер массива должен быть передан в качестве дополнительного аргумента. Один массив не может быть присвоен другому массиву через операцию присваивания. Подобные свойства и действия выглядели бы «естественно» при работе с массивами, но в C и C++ отсутствует их реализация. Однако в C++ имеется возможность реализовать подобные свойства массивов посредством механизма перегрузки операций.

В следующем примере мы разработаем класс массива, который выполняет проверку индексации, чтобы убедиться, что она остается в заданных границах массива. Этот класс позволяет использовать операцию присваивания для присваивания одного объекта-массива другому. Объекты массива этого класса автоматически знают свой размер, поэтому нет необходимости передавать его в качестве аргумента при передаче массива функции. Весь массив в целом может быть введен или выведен при помощи операций извлечения из потока и операции передачи в поток. Сравнение массивов может осуществляться при помощи операций == или !=. Наш класс массива использует статические элементы, чтобы отслеживать число объектов-массивов, которые были созданы в программе. Этот пример улучшит ваше понимание абстрактных данных. Вероятно, вы сможете предложить много расширений такого класса массива. Разработка класса — это интересная задача, требующая немалых интеллектуальных усилий.

Программа, приведенная на рис. 18.4, демонстрирует класс **Array** и его перегруженные операции. Сначала мы пройдемся по программе-тестеру в **main**. Затем мы рассмотрим определение класса, определения его функций-элементов и определения дружественных функций. Программа создает два объекта класса **Array** — **integers1** с семью элементами и **integers2**, имеющий размер 10 элементов по умолчанию (значение по умолчанию определено конструктором **Array**). Статический элемент данных **arrayCount** класса **Array** содержит число объектов **Array**, созданных во время выполнения программы. Статическая функция-элемент **getArrayCount** возвращает это значение. Функция-элемент **getSize** возвращает размер массива **integers1**. Программа выводит размер массива **integers1** и выводит сам массив, используя перегруженную операцию передачи в поток для подтверждения того, что элементы массива были корректно инициализированы конструктором. Затем выводится размер массива **integers2** и выводится сам массив с помощью перегруженной операции передачи в поток.

Пользователю предлагается ввести 17 целых. Для чтения этих значений в оба массива используется перегруженная операция извлечения из потока, выполняемая оператором

```
cin >> integers1 >> integers2;
```

Первые семь значений сохраняются в **integers1**, а остальные значения в **integers2**. Чтобы подтвердить корректность ввода, эти два массива выводятся операцией передачи в поток.

Далее программа проверяет перегруженную операцию неравенства вычислением условия

```
integers1 != integers2
```

и сообщает, что два массива действительно не равны.

```
// ARRAY1.H
// Простой класс Array (для целых)
#ifndef array1_h
#define ARRAY1_H

#include <iostream.h>

class Array {
   friend ostream &operator<<(ostream &, const Array &);
   friend istream &operator>>(istream &, Array &);
public:
   Array(int = 10);                    // конструктор по умолчанию
   Array(const Array &);               // конструктор копии
   ~Array();                           // деструктор
   int getSize() const;                // возврат size
   const Array &operator=(const Array &); // присваивание
                                          // массивов
   int operator==(const Array &) const; // сравнение на равенство
   int operator!=(const Array &) const; // сравнение
                                        // на !равенство
   int &operator[](int);                // операция индексации
   static int getArrayCount();          // возврат числа
                                        // созданных массивов
private:
   int *ptr; // указатель на первый элемент массива
   int size; // размер массива
   static int arrayCount; // число созданных массивов
};

#endif
```

Рис.18.4. Определение класса **Array** (часть 1 из 7)

```
// ARRAY1.CPP
// Определения функций-элементов класса Array
#include <iostream.h>
#include <stdlib.h>
#include <assert.h>
#include "array1.h"

// Инициализация статических данных в области действия файла
int Array::arrayCount = 0;   // пока объектов нет

// Возврат числа созданных объектов Array
int Array::getArrayCount() { return arrayCount; }

// Конструктор по умолчанию для класса Array
Array::Array(int arraySize)
{
   ++arrayCount;              // приращение счетчика на один объект
   size = arraySize;          // размер по умолчанию - 10
   ptr = new int[size];       // создать свободное место для массива
   assert(ptr != 0);          // выход, если размещения памяти
                              // не произошло
```

Рис. 18.4. Определения функций-элементов класса **Array** (часть 2 из 7)

```
   for (int i = 0; i < size; i++)
      ptr[i] = 0;              // инициализировать массив
}

// Конструктор копии для класса Array
Array::Array (const Array &init)
{
   ++arrayCount;             // приращение счетчика на один объект
   size = init.size;         // размер этого объекта
   ptr = new int [size];     // создать свободное место для массива
   assert(ptr != 0);         // выход, если памяти не размещена

   for (int i = 0; i < size; i++)
      ptr[i] = init.ptr[i];  // скопировать init в объект
}

// Деструктор для класса Array
Array::~Array()
{
   --arrayCount;             // одним объектом меньше
   delete [] ptr;            // восстановить место для массива
}

// Получить размер массива
int Array::getSize() const { return size; }

// Перегруженная операция индексации
int &Array::operator[](int subscript)
{
   // проверка выхода индекса за пределы массива
   assert(0 <= subscript && subscript < size);

   return ptr[subscript];   // возврат ссылки создает lvalue
}

// Определение равенства двух массивов и возврат
// 1, если условие выполнено и 0, если не выполнено.
int Array::operator==(const Array &right) const
{
   if (size != right.size)
      return 0;      // массивы различных размеров

   for (int i = 0; i < size; i++)
      if (ptr[i] != right.ptr[i])
         return 0; // массивы не равны

   return 1;        // массивы равны
}

// Определение неравенства двух массивов и возврат
// 1, если условие выполнено и 0, если не выполнено.
int Array::operator!=(const Array &right) const
{
   if (size != right.size)
      return 1;     // массивы различных размеров
```

Рис. 18.4. Определения функций-элементов класса **Array** (часть 3 из 7)

```
   for (int i = 0; i < size; i++)
      if (ptr[i] != right.ptr[i])
         return 1;       // массивы не равны

   return 0;         // массивы равны
}

// Перегруженная операция присваивания
const Array &Array::operator=(const Array &right)
{
   if (&right != this) {    // проверка на самоприсваивание
      delete [] ptr;        // восстановить свободное место
      size = right.size;    // изменить размеры этого объекта
      ptr = new int[size]; // создать место для копии массива
      assert(ptr != 0);     // выход, если память не размещена

      for (int i = 0; i < size; i++)
         ptr[i] = right.ptr[i];  // скопировать массив в объект
   }

   return *this;            // разрешить x=y=z;
}

// Перегруженная операция ввода для класса Array;
// ввод значений для всего массива.
istream &operator>>(istream &input, Array &a)
{
   for (int i=0; i< a.size; i++)
      input >> a.ptr[i];

   return input;            // разрешить cin >> x >> y;
}

// Перегруженная операция вывода для класса Array
ostream &operator<<(ostream &output, const Array &a)
{
   for (int i = 0; i < a.size; i++) {
      output << a.ptr[i] << ' ';

      if ((i+1) % 10 == 0)
         output << endl;
   }

   if (i % 10 != 0)
      output << endl;

   return output;    // разрешить cout << x << y;
}
```

Рис. 18.4 Определения функций-элементов класса **Array** (часть 4 из 7)

Программа создает третий массив, названный **integers3**, и инициализирует его массивом **integers1**. Программа выводит размер массива **integers3** и выводит сам массив, используя перегруженную операцию передачи в поток, чтобы подтвердить, что элементы массива были корректно инициализированы конструктором.

```
// FIG18_4.CPP
// Тестер для простого класса Array
#include <iostream.h>
#include "array1.h"

main( )
{
   // пока объектов нет
   cout << "# of arrays instantiated = "
        << Array::getArrayCount() << '\n';

   // создать два массива и вывести значение счетчика Array
   Array integers1(7), integers2;
   cout << "# of arrays instantiated = "
        << Array::getArrayCount() ;

// вывести размер integers1 и содержимое
cout << "\n\nSize of array integers1 is "
     << integers1.getSize()
     << "\nArray after initialization:\n"
     << integers1;

// вывести размер integers2 и содержимое
cout << "\nSize of array integers2 is "
     << integers2.getSize()
     << "\nArray after initialization:\n"
     << integers2;

// ввод и вывод integers1 и integers2
cout << "\nInput 17 integers:\n";
cin >> integers1 >> integers2;
cout << "After input, the arrays contain:\n"
     << "integers1: " << integers1
     << "integers2: " << integers2;

// создать массив integers3, используя integers1 в качестве
// инициализатора; вывести размер и содержимое
Array integers3 (integers1);

cout << "\nSize of array integers3 is "
     << integers3.getSize()
     << "\nArray after initialization: \n"
     << integers3;

// использовать перегруженную операцию неравенства(!=)
cout << "\nEvaluating: integers1 != integers2\n";
if (integers1 != integers2)
   cout << "They are not egual\n\n";

// использовать перегруженную операцию присваивания (=)
cout << "Assigning integers2 to integers1: \n";
integers1 = integers2;
cout << "integers1: " << integers1
     << "integers2: " << integers2;
```

Рис. 18.4 Тестер для класса **Array** (часть 5 из 7)

```
// использовать перегруженную операцию равенства(==)
cout << "\nEvaluating: integers1 == integers2\n";
if (integers1 == integers2)
   cout << "They are equal\n\n";

// использовать перегруженную операцию индексации
//для создания rvalue
cout << "integers1[5] is " << integers1[5] << endl;

// использовать перегруженную операцию индексации
// для создания lvalue
cout << "Assigning 1000 to integers1[5] \n";
integers1[5] = 1000;
cout << "integers1: " << integers1;

// попытка использовать индекс, выходящий за размеры массива
cout << "\nAttempt to assign 1000 to integers1[15] \n";
integers1[15] = 1000;  // ОШИБКА: выход за границы массива

return 0;
}
```

Рис. 18.4 Тестер для класса **Array** (часть 6 из 7)

Затем программа проверяет перегруженную операцию присваивания (=) посредством выражения **integers1 = integers2**. Оба массива после этого выводятся на печать для подтверждения правильности присваивания. Интересно отметить, что **integers1** содержит изначально 7 целых и требуется изменение размера этого массива, чтобы он сохранял копию 10 элементов из массива **integers2**. Как мы увидим, перегруженная операция присваивания выполняет изменение размера массива скрытым от вызвавшей операцию программы образом.

Затем программа применяет перегруженную операцию равенства (==), чтобы убедиться, что объекты **integers1** и **integers2** после выполнения операции присваивания действительно идентичны.

Далее программа использует перегруженную операцию индексации для ссылки на элемент **integers1[5]**, находящийся в допустимых пределах **integers1**. Это индексированное имя используется как *rvalue* для печати значения **integers1[5]** и как *lvalue* с левой стороны операции присваивания, чтобы присвоить новое значение — 1000 — элементу **integers1[5]**. Обратите внимание, что **operator[]** возвращает ссылку.

После этого программа пытается присвоить значение 1000 элементу **integers1[15]**, который выходит за границы массива. Программа перехватывает ошибку и прекращает работу аварийно.

Интересно, что операцию индексации **[]** не запрещено перегружать и для других целей: она может использоваться для выбора элементов из других видов контейнерных классов, таких, как связанные списки, строки, словари и т.д. Индексам также не обязательно быть только целыми, например, они могут быть символами или строками.

Теперь, когда мы разобрались с работой главной программы, давайте пройдемся по заголовкам классов и определениям функций-элементов. Строки программы

```
int *ptr; // указатель на первый элемент массива
int size; // размер массива
static int arrayCount;// число созданных массивов
```

объявляют закрытые элементы данных. Массив содержит указатель **ptr** (ссылается на соответствующий тип, в данном случае **int**), элемент **size**, показывающий число элементов в массиве, и статический элемент **arrayCount**, показывающий число созданных объектов-массивов.

```
# of arrays instantiated = 0
# of arrays instantiated = 2

Size of array integers1 is 7
Array after initialization:
0 0 0 0 0 0 0

Size of array integers2 is 10
Array after initialization:
0 0 0 0 0 0 0 0 0 0

Input 17 integers:
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17
After input, the arrays contain;
integers1: 1 2 3 4 5 6 7
integers2: 8 9 10 11 12 13 14 15 16 17

Size of array integers3 is 7
Array after initialization;
1 2 3 4 5 6 7

Evaluating: integers1 != integers2
They are not equal

Assigning integers2 to integers1:
integers1: 8 9 10 11 12 13 14 15 16 17
integers2: 8 9 10 11 12 13 14 15 16 17

Evaluating: integers1 == integers2
They are equal

integers1[5] is 13
Assigning 1000 to integers1[5]
integers1: 8 9 10 11 12 1000 14 15 16 17

Attempt to assign 1000 to integers1[15]
Assertion failed: 0 <= subscript && subscript < size,
file ARRAY1.CPP, line 98
Abnormal program termination
```

Рис. 18.4 Вывод программы-тестера для класса **Array** (часть 7 из 7)

Строки программы

```
friend ostream &operator<<(ostream &, const Array &);
friend istream &operator>>(istream &, Array &);
```

объявляют перегруженные операции передачи в поток и извлечения из потока как друзей класса **Array**. Когда компилятор встречает выражение, аналогичное

```
cout << arrayObject
```

он вызывает функцию **operator<<**, генерируя вызов

```
operator<<(cout, arrayObject)
```

Когда компилятор встречает выражение, аналогичное

```
cin >> arrayObject
```

он вызывает функцию **operator>>**, генерируя вызов

```
operator>>(cin, arrayObject)
```

Мы еще раз отметим, что эти функции-операции не могут быть элементами класса **Array**, так как объект, принадлежащий **Array**, всегда находится с правой стороны операций передачи и извлечения из потока. Функция **operator<<** выводит число элементов, определяемое **size**, из массива, хранящегося по адресу **ptr**. Функция **operator>>** осуществляет ввод данных прямо в массив, хранящийся по адресу **ptr**. Каждая их этих функций-операций возвращает соответствующую ссылку, чтобы сделать возможной конкатенацию вызовов.

Строка программы

```
Array(int arraySize = 10); // конструктор по умолчанию
```

объявляет конструктор класса по умолчанию и определяет размер массива по умолчанию (10 элементов). Когда компилятор встречает такое объявление, как

```
Array integers1(7);
```

или эквивалентную запись

```
Array integers1 = 7;
```

он вызывает конструктор по умолчанию. Конструктор по умолчанию — функция-элемент **Array** — действует следующим образом: увеличивает значение счетчика **arrayCount**; копирует аргумент в **size** (элемент данных); использует **new** для выделения места под хранение внутреннего представления этого массива и присваивает **ptr** значение указателя, возвращаемого **new**; вызывает **assert** для проверки успешности вызова **new**; организует цикл **for** для инициализации всех элементов массива значением 0. Было бы также вполне допустимым определить такой класс **Array**, который не проводит инициализацию своих элементов, если, например, требуется, чтобы эти элементы считывались позднее.

Строка программы

```
Array(const Array &);// конструктор копии
```

является *конструктором копии*. Он инициализирует объект **Array** копированием существующего объекта **Array**. Такое копирование должно выполняться осторожно, чтобы избежать ловушки получения двух объектов **Array**, указывающих на одну и ту же динамически выделенную память, т.е. здесь та же самая проблема, что и при поэлементном копировании элементов данных по умолчанию. Конструктор копии вызывается всякий раз, когда необходимо копирование объекта: при вызове по значению, при возврате объекта из вызванной функции; при инициализации, когда один объект копируется в другой объект того же класса. Конструктор копии вызывается при объявлении, когда создается объект класса **Array** и инициализируется другим объектом класса **Array**, как в следующем объявлении:

```
Array integers3 (integers1);
```

или эквивалентном ему

```
Array integers3 = integers1;
```

Конструктор копии — функция-элемент **Array** — увеличивает значение счетчика **arrayCount**; копирует из массива значение **size**, использующееся для *инициализации массива*, *в элемент данных* **size**; использует **new** для получения свободного места под хранение внутреннего представления этого массива и присваивает **ptr** значение указателя, возвращаемого **new**; использует **assert** для проверки успешности вызова **new**; использует цикл **for** для копирования всех элементов массива-инициализатора в этот массив. Важно отметить, что, если конструктор копии просто копирует **ptr** объекта-источника в **ptr** создаваемой копии, то оба объекта могут указывать на одну и ту же область динамически размещаемой памяти. Первый вызов деструктора должен был бы после этого очистить динамически выделенную память, и другие объекты, на которые ссылается **ptr**, стали бы неопределенными. Такая ситуация чаще всего вызывает серьезную ошибку при выполнении программы.

Хороший стиль программирования 18.6

Деструктор, операция присваивания и конструктор копии класса обычно реализуются как единая группа.

Строка программы

```
~Array();   // деструктор
```

объявляет деструктор класса. Деструктор вызывается автоматически, когда существование объекта класса **Array** заканчивается. Деструктор уменьшает значение счетчика **arrayCount** и использует **delete** для освобождения динамической памяти, выделенной **new** в конструкторе.

Строка программы

```
int getSize() const;// возвращаемый размер
```

объявляет функцию, которая читает размер массива.

Строка программы

```
const Array &operator=(const Array &); //присваивание массива
```

объявляет перегруженную функцию-операцию присваивания класса. Когда компилятор встречает выражение

```
integers1 = integers2
```

он вызывает функцию **operator=**, генерируя вызов

```
integers1.operator=(integers2)
```

Функция-элемент **operator=** производит проверку на самоприсваивание. Если делается попытка самоприсваивания, операция пропускается (т.е. если объект уже является сам собой). Если это не самоприсваивание, тогда функция-элемент использует **delete** для освобождения памяти, первоначально отведенной под массив-приемник; копирует из массива-источника значение **size** в **size** массива-копии; использует **new** для выделения места под массив-копию и присваевает указатель, возвращаемый **new**, элементу **ptr** приемника; вызывает **assert** для проверки успешности вызова **new**; использует цикл **for** для копирования элементов массива-источника в массив-приемник. Вне зависимости от того, самоприсваивание это или нет, функция-элемент возвращает *после* вызова свой объект (т.е. ***this**) как константную ссылку; это разрешает конкатенацию присваиваний **Array**, таких, как **x=y=z**. Если бы контроль самоприсваивания был пропущен, функция-элемент не могла бы начинаться с освобождения памяти массива-приемника. Поскольку при самоприсвоении он также и *массив-источник*, этот массив не должен быть разрушен.

Распространенная ошибка программирования 18.4

Отсутствие проверки и обхода самоприсваивания при перегрузке операции присваивания класса, который содержит указатель на динамическую память.

Распространенная ошибка программирования 18.5

Отсутствие перегруженной операции присваивания и конструктора копии класса, когда объекты этого класса содержат указатели на динамическую размещаемую память.

Общее методическое замечание 18.3

Имеется возможность избежать присваивания одного объекта класса другому. Это можно сделать, определив операцию присваивания как закрытый элемент класса.

Строка программы

```
int operator==(const Array &) const; // сравнение на равенство
```

объявляет перегруженную операцию равенства (==) класса. Когда компилятор встречает в **main** выражение

```
integers1 == integers2
```

он вызывает функцию-элемент **operator==**, генерируя вызов

```
integers1.operator==(integers2)
```

Функция-операция **operator==** сразу возвращает 0 (false), если размеры элементов массива **size** различны. В противном случае функция-элемент сравнивает каждую пару элементов. Если все они одинаковы, возвращается 1 (true). Первая пара отличающихся элементов сразу приводит к возврату 0.

Строка программы

```
int operator!=(const Array &) const; // сравнение на !равенство
```

объявляет перегруженную операцию неравенства (!=) класса. Вызывается функция-элемент **operator!=**, которая работает аналогично перегруженной функции-операции проверки на равенство.

Строка программы

```
int operator[](int);//операция индексации
```

объявляет перегруженную операцию индексации класса. Когда компилятор встречает в **main** выражение

```
integers1[5]
```

он вызывает функцию-элемент **operator[]**, генерируя вызов

```
integers1.operator[](5)
```

Функция-операция **operator[]** проверяет, находится ли индекс в допустимых пределах и, если нет, программа аварийно завершается. Если индекс имеет допустимое значение, возвращается соответствующий элемент массива как ссылка, которая может использоваться как *lvalue* (например, с левой стороны операции присваивания).

Строка программы

```
static int getArrayCount(); // возврат счетчика массивов
```

объявляет статическую функцию-элемент **getArrayCount**, которая возвращает значение статического элемента данных **arrayCount**, даже если не существует объектов класса **Array**.

18.9. Преобразование типов

Большинство программ обрабатывают информацию разных типов. Иногда все операции «остаются в том же типе». Например, суммирование целого с целым дает целое (до тех пор, пока результат не становится слишком большим, когда он уже не может быть представлен целым значением). Однако часто возникает необходимость преобразования данных одного типа в данные другого типа. Это может происходить при присваивании, в вычислениях, при передаче значений функциям и при возврате значений функциями. Компилятор знает, как выполнять преобразование для встроенных типов. Программист может управлять преобразованиями встроенных типов посредством приведения типов.

Но как насчет типов, определяемых пользователем? Компилятор не может автоматически знать, как проводить преобразования между типами, определяемыми пользователем, и встроенными типами. Программист должен определить в явном виде, каким образом проводить такие преобразования. Преобразования такого рода могут проводиться с использованием *конструктора преобразований*, т.е. конструктора с одним аргументом, который просто превращает объект некоторого типа в объект данного класса. Мы будем использовать конструктор преобразования позже в этой главе для преобразования обычных строк символов **char *** в объекты класса C++ **String**.

Операция преобразования (также называемая *операцией приведения*) может использоваться для преобразования объекта одного класса в объект другого класса или в объект встроенного типа. Такого рода операция преобразования должна быть не-статической функцией-элементом; этот вид операции не может быть функцией-другом.

Прототип функции

```
operator char *() const;
```

объявляет перегруженную функцию-операцию приведения для создания временного объекта **char ***, *не принадлежащего типу объекта, определяемого пользователем*. Перегруженная функция-операция приведения не специфицирует возвращаемый тип (т.е. тот, к которому преобразуется данный объект). Когда компилятор встречает выражение **(char *) s**, он генерирует вызов **s.operator char *()**. Операнд **s** является объектом класса **s**, для которого вызывается функция-элемент **operator char ***.

Перегруженная функция-операция приведения может определяться с целью преобразования объектов типов, определяемых пользователем, в объекты встроенных типов или в объекты других типов, созданных пользователем. Прототипы функций

```
operator int() const;
operator otherClass() const;
```

объявляют перегруженные функции-операции приведения для преобразования объектов типа, определяемого пользователем, в целое, и для преобразования объектов типа, определяемого пользователем, в объекты типа **otherClass**, также созданного пользователем.

Одной их замечательных возможностей операций приведения и конструкторов преобразования является то, что при необходимости компилятор может автоматически вызывать эти функции, чтобы создавать временные объекты. Например, если объект **s** класса **String**, определенного пользователем, появляется в программе в том месте, где предполагается обычная строка **char ***, например

```
cout << s;
```

то компилятор вызывает перегруженную функцию-операцию приведения **operator char ***, чтобы преобразовать этот объект в **char *** и использовать в выражении полученную строку.

18.10. Пример: класс String

В целях исследования перегрузки операций мы построим класс, который управляет созданием и обработкой строк. Ни в C, ни C++ не имеется встроенного типа строковых данных. Но C++ позволяет нам ввести свой собственный тип в качестве класса и, посредством механизма перегрузки, определить набор операций для удобной обработки строк. На рис. 18.5 показаны различные части класса **String**: заголовок класса, определения функций-элементов и программа-тестер для проверки класса.

Сначала мы рассмотрим заголовок класса **String**. Мы обсудим закрытые данные, использующиеся для представления объектов **String**. Затем мы пройдемся по открытому интерфейсу класса, обсуждая каждый вид услуг, предоставляемых классом.

После этого мы рассмотрим программу-тестер в **main**. Мы обсудим стиль программирования, к которому мы стремимся, т.е. к насыщенным операциями выражениям, которые мы могли бы составить из объектов нашего класса **String** и набора перегруженных операций.

Затем мы разберем определения функций-элементов класса **String**. Для каждой из перегруженных операций мы приведем программу-тестер, которая вызывает перегруженную функцию-операцию и объясним, как она работает.

Мы начнем со внутреннего представления **String**. Строки программы

```
private:
   char *sPtr;    // указатель на начало строки
   int length;    // длина строки
```

объявляют закрытые элементы данных класса. Объект **String** имеет указатель на динамически распределяемую память для представления строки символов и поле длины, которое представляет число символов в строке, исключая ограничивающий строку нулевой символ.

Теперь перейдем к заголовочному файлу класса **String** (рис. 18.5). Строки программы

```
friend ostream &operator<<(ostream &, const String &);
friend istream &operator>>(istream &, String &);
```

объявляют перегруженную функцию-операцию передачи в поток **operator<<** и перегруженную функцию-операцию извлечения из потока **operator>>** как друзей класса. Их применение очевидно.

Строка программы

```
String(const char * = ""); // конструктор преобразования
```

объявляет *конструктор преобразования*. Этот конструктор принимает в качестве аргумента **char *** (который является по умолчанию пустой строкой) и создает объект **String**, который содержит ту же самую строку символов. Любой конструктор с единственным аргументом можно рассматривать в качестве конструктора преобразования. Как мы увидим, такой конструктор полезен, когда мы присваиваем объекту **String** строку символов. Конструктор преобразует обычную строку в объект **String**, который, в свою очередь, присваивается объекту

String. Наличие такого конструктора преобразования означает, что нет необходимости предусматривать специальную операцию присваивания для присвоения объекту **String** строки символов. Компилятор автоматически вызывает конструктор преобразования, чтобы создать временный объект **String**, содержащий указанную строку. После этого вызывается перегруженная операция присваивания, чтобы присвоить временный объект **String** другому объекту **String**. Обратите внимание, что когда C++ использует конструктор преобразования таким образом, он может применить лишь один конструктор, чтобы попробовать удовлетворить потребности перегруженной операции присваивания. Наоборот, невозможно реализовать такое присваивание посредством выполнения ряда неявных преобразований, определенных пользователем. Конструктор преобразования **String** может вызываться в таком объявлении, как **String s1("happy")**. Конструктор преобразования вычисляет длину строки символов и присваивает это значение закрытому элементу данных **length**; использует **new**, чтобы присвоить указатель на выделенную под строку память закрытому элементу данных **sPtr**; вызывает **assert** для проверки успешности вызова **new** и, если память выделена, вызывает **strcpy**, чтобы скопировать строку символов в объект.

```
// STRING2.H
// Определение класса String
#ifndef STRING1_H
#define STRING1_H

#include <iostream.h>

class String {
   friend ostream &operator<<(ostream &, const String &);
   friend istream &operator>>(istream &, String &);

public:
   String(const char * = ""); // конструктор преобразования
   String(const String &);    // конструктор копии
   ~String();                 // деструктор
   const String &operator=(const String &); // присваивание
   String &operator+=(const String &);    // конкатенация
   int operator!() const;                 // String пустая?
   int operator==(const String &) const;  // проверка s1 == s2
   int operator!=(const String &) const;  // проверка s1 != s2
   int operator<(const String &)  const;  // проверка s1 < s2
   int operator>(const String &)  const;  // проверка s1 > s2
   int operator>=(const String &) const;  // проверка s1 >= s2
   int operator<=(const String &) const;  // проверка s1 <= s2
   char &operator[](int);          // возврат ссылки char
   String &operator()(int, int); // возврат подстроки
   int getLength() const;        // возврат длины строки

private:
   char *sPtr;                   // указатель на начало строки
   int length;                   // длина строки
};

#endif
```

Рис. 18.5. Определение класса **String** (часть 1 из 7)

```
// STRING2.CPP
// Определения функций-элементов для класса String

#include <iostream.h>
#include <string.h>
#include <assert.h>
#include "string2.h"

// Конструктор преобразования: Преобразует char * в String
String::String(const char *s)
{
   cout << "Conversion constructor: " << s << endl;
   length = strlen(s);              // вычислить длину
   sPtr = new char[length + 1];     // выделить память
   assert(sPtr != 0);               // выход, если не выделена
   strcpy(sPtr, s);                 // скопировать строку в объект
}

// Конструктор копии
String::String(const String &copy)
{
   cout << "Copy constructor: " << copy.sPtr << endl;
   length = copy.length;            // скопировать длину
   sPtr = new char[length + 1];     // выделить память
   assert(sPtr != 0);               // проверить результат
   strcpy(sPtr, copy.sPtr);         // скопировать строку
}

// Деструктор
String::~String()
{
   cout << "Destructor: " << sPtr << endl;
   delete [] sPtr;// очистить строку
}

// Перегруженная операция =; избегать самоприсваивания
const String &String::operator=(const String &right)
{
   cout << "operators called" << endl;

   if (&right != this) {            // избегать самоприсваивания
      delete [] sPtr;               // предотвратить утечку памяти
      length = right.length;        // новая длина строки
      sPtr = new char[length +1]; // выделить память
      assert (sPtr != 0);           // проверка результата
      strcpy(sPtr, right.sPtr);     // скопировать строку
   }
   else
      cout << "Attempted assignment of a String to itself \n";

   return *this;   // разрешить конкатенацию присваиваний
}

// Конкатенация правого операнда с этим объектом и
// сохранение результата в этом объекте.
String &String::operator+=(const String &right)
```

Рис. 18.5 Определения функций-элементов для класса **String** (часть 2 из 7)

```
{
   char *tempPtr = sPtr; // сохранить, чтобы иметь возможность
                         //стереть
   length += right.length;      // новая длина String
   sPtr = new char[length +1]; // создать свободное место
   assert(sPtr != 0);           // выйти, если память не размещена
   strcpy(sPtr, tempPtr);       // левая часть новой String
   strcat(sPtr, right.sPtr);    // правая часть новой String
   delete [] tempPtr;           // восстановить старое место
   return *this;                // разрешить конкатенацию вызовов
}

// String пустая?
int String::operator!() const { return length == 0; }

// String равна правой String?
int String::operator==(const String &right) const
{ return strcmp(sPtr, right.sPtr) == 0; }

// String не равна правой String?
int String::operator!=(const String &right) const
{ return strcmp(sPtr, right.sPtr) != 0; }

// String меньше чем правая String?
int String::operator<(const String &right) const
{ return strcmp(sPtr, right.sPtr) < 0; }

// String больше чем правая String?
int String::operator>(const String &right) const
{ return strcmp(sPtr, right.sPtr) > 0; }

// String больше чем или равна правой String?
int String::operator>=(const String &right) const
{ return strcmp(sPtr, right.sPtr) >= 0; }

// String меньше чем или равна правой String?
int String ::operator<=(const String &right) const
{ return strcmp (sPtr, right.sPtr) <= 0; }

// Возврат ссылки на символ в String.
char &String::operator [](int subscript)
{
   // Сначала проверяем выход индекса за границы
   assert(subscript >= 0 && subscript < length);

   return sPtr [subscript];  // создать lvalue
}

// Возврат подстроки, начинающейся с index и
// длиной subLength как ссылки на объект String.
String &String::operator()(int index, int subLength)
{
   // убедимся, что index находится в границах
   // и длина подстроки >= 0
   assert(index >= 0 && index < length && subLength >= 0);
```

Рис. 18.5 Определения функций-элементов для класса **String** (часть 3 из 7)

```
   String *subPtr = new String;    // пустая String
   assert(subPtr != 0);            // проверка выделения памяти

   // определение длины подстроки
   int size;

   if ((subLength == 0) || (index + subLength > length))
      size = length - index + 1;
   else
      size = subLength + 1;

   // разместить память для подстроки
   delete subPtr->sPtr;
   subPtr->length = size;
   subPtr->sPtr = new char[size];
   assert(subPtr->sPtr != 0);      // проверка выделения памяти

   // скопировать подстроку в новую String
   for (int i = index, j = 0; i < index + size - 1; i++, j++)
      subPtr->sPtr[j] = sPtr[i];

   subPtr->sPtr[j] = '\0';         // ограничить новую строку
                                   // нулевым символом
   return *subPtr;                 // возврат новой String
}

// Возврат длины строки
int String::getLength() const { return length; }

// Перегруженная операция вывода
ostream &operator<<(ostream &output, const String &s)
{
   output << s.sPtr;
   return output;    // разрешить конкатенацию
}

// Перегруженная операция ввода
istream &operator>>(istream &input, String &s)
{
   static char temp[100];    // буфер ввода

   input >> temp;
   s = temp;          // используем операцию присваивания
                      // класса String
   return input;      // разрешить конкатенацию
}
```

Рис. 18.5 Определения функций-элементов для класса **String** (часть 4 из 7)

Строка программы

```
String(const String &);     // конструктор копии
```

является конструктором копии. Он инициализирует объект **String** копированием существующего объекта. Такое копирование должно выполняться аккуратно, чтобы избежать ловушки, когда в результате копирования будут получены два объекта **String**, указывающих на одну и ту же динамически выделенную память, — т.е. тот же случай, который может произойти при поэле-

ментном копировании по умолчанию. Конструктор копии работает аналогично конструктору преобразования за исключением того, что он просто копирует элемент **length** из исходного объекта класса **String** в объект-копию **String**. Обратите внимание, что конструктор копии заново выделяет память для внутренней строки символов объекта-копии. Если бы он просто скопировал **sPtr** из исходного объекта в **sPtr** объекта-копии, то оба объекта указывали бы на ту же самую область памяти. При первом вызове деструктора динамически выделенная память была бы очищена и **sPtr** другого объекта стал бы неопределенным. Такая ситуация с большой вероятностью вызовет серьезную ошибку при выполнении программы.

```
// FIG18_5.CPP
// Тестер для класса String
#include <iostream.h>
#include "string2.h"

main()
{
   String s1("happy"), s2(" birthday"), s3;

   // проверка перегруженных операций равенства и отношений
   cout << "s1 is \"" << s1 << "\"; s2 is \"" << s2
        << "\"; s3 is empty\n"
        << "The results of comparing s2 and s1:"
        << "\ns2 == s1 yields " << (s2 == s1)
        << "\ns2 != s1 yields " << (s2 != s1)
        << "\ns2 >  s1 yields " << (s2 > s1)
        << "\ns2 <  s1 yields " << (s2 < s1)
        << "\ns2 >= s1 yields " << (s2 >= s1)
        << "\ns2 <= s1 yields " << (s2 <= s1) << "\n\n";

   // проверка содержимого String перегруженной операцией(!)
   cout << "Testing !s3:\n";
   if (!s3) {
      cout << "s3 is empty; assigning s1 to s3;\n";
      s3 = s1;    // проверка перегруженной операции присваивания
      cout << "s3 is \"" << s3 << "\"\n\n";
   }

   // проверка перегруженной операции конкатенации String
   cout << "s1 += s2 yields s1 = ";
   s1 += s2;              // проверка перегруженной конкатенации
   cout << s1 << "\n\n";

   // проверка перегруженного конструктора
   cout << "s1 += \" to you\" yields\n";
   s1 += " to you";       // проверка перегруженного конструктора
   cout << "s1 " << s1 << "\n\n";

   // проверка подстроки перегруженной
   // функцией-операцией вызова ()
   cout << "The substring of s1 starting at\n"
        << "location 0 for 14 characters, s1(0, 14), is: "
        << s1(0, 14) << "\n\n";
```

Рис. 18.5 Тестер для проверки класса **String** (часть 5 из 7)

```
   // проверка опции подстроки (до конца строки)
   cout << "The substring of s1 starting at\n"
        << "location 15, s1(15, 0), is: "
        << s1(15, 0) << "\n\n";  // 0 означает "до конца строки"

   // проверка конструктора копии
   String *s4Ptr = new String(s1);
   cout << "*s4Ptr = " << *s4Ptr << "\n\n";

   // проверка операции присваивания (=) с самоприсваиванием
   cout << "assigning *s4Ptr to *s4Ptr\n";
   *s4Ptr = *s4Ptr;   // проверка перегруженного самоприсваивания
   cout << "*s4Ptr = " << *s4Ptr << endl;

   // проверка деструктора
   delete s4Ptr;

   // проверка операции индексирования, возвращающей lvalue
   s1[0] = 'H';
   s1[6] = 'B';
   cout << "\ns1 after s1[0] = 'H' and s1[6] = 'B' is: "
        << s1 << "\n\n";

   // проверка на выход индекса из границ
   cout << "Attempt to assign 'd' to s1[30] yields:\n";
   s1[30] = 'd';      // ОШИБКА: индекс вышел за границы

   return 0;
}
```

Рис. 18.5 Тестер для проверки класса **String** (часть 6 из 7)

Строка программы

```
~String();// деструктор
```

объявляет деструктор для класса **String**. Деструктор применяет операцию **delete**, чтобы очистить динамическую память, полученную **new** в конструкторе.

Строка программы

```
const String &operator=(const String &);     // присваивание
```

объявляет перегруженную операцию присваивания. Когда компилятор встречает такое выражение, как **string1 = string2**, он вызывает функцию

```
string1.operator=(string2);
```

Перегруженная функция-операция присваивания **operator=** проверяет операнды на самоприсваивание. Если регистрируется самоприсваивание, то следует просто возврат из функции, потому что сам объект уже и является требуемым объектом. Если бы эта проверка была опущена, то функция немедленно очистила бы **sPtr** в объекте и строка символов была бы потеряна. Если это не самоприсваивание, функция особождает старую строку и копирует поле длины исходного объекта в объект-копию; заново выделяет память для строки-копии; вызывает **assert** для проверки успешности вызова **new** и после этого вызывает **strcpy**, чтобы скопировать строку символов из исходного объекта в объект-копию. Вне зависимости от того, самоприсваивание это или нет, функция возвращает ***this**, чтобы была возможность конкатенации присвоений.

```
Conversion constructor: happy
Conversion constructor:  birthday
Conversion constructor:
s1 is "happy"; s2 is " birthday"; s3 is empty
The results of comparing s2 and s1:
s2 == s1 yields 0
s2 != s1 yields 1
s2 >  s1 yields 0
s2 <  s1 yields 1
s2 >= s1 yields 0
s2 <= s1 yields 1

Testing !s3:
s3 is empty; assigning s1 to s3;
operator= called
s3 is "happy"

s1 += s2 yields s1 = happy birthday

s1 += " to you" yields
Conversion constructor:  to you
Destructor:  to you
s1 = happy birthday to you

Conversion constructor:
The substring of s1 starting at
location 0 for 14 characters, s1(0, 14), is: happy birthday

Conversion constructor:
The substring of s1 starting at
location 15, s1 (15, 0), is: to you

Copy constructor; happy birthday to you
*s4Ptr = happy birthday to you

assigning *s4Ptr to *s4Ptr
operator= called
Attempted assignment of a String to itself
*s4Ptr = happy birthday to you
Destructor: happy birthday to you

s1 after s1(0) == 'H' and s1[6] = 'B' is: Happy Birthday to you

Attempt to assign 'd' to s1[30] yields:
Assertion failed: subscript >= 0 && subscript < length,
   file STRING2.CPP, line 98
Abnormal program termination
```

Рис.18.5 Вывод программы-тестера для проверки класса (часть 7 из 7)

Строка программы

```
String &operator+=(const String &);     // конкатенация
```

объявляет перегруженную операцию конкатенации строк. Когда компилятор встречает в **main** выражение **s1 += s2**, он генерирует вызов функции **s1.operator+=(s2)**. Функция **operator+=** создает временный указатель для хранения

текущего указателя строки символов объекта до того момента, когда становится возможным очистить память, содержащую строку символов; вычисляет суммарную длину двух строк; использует **new**, чтобы зарезервировать место для строки результата; вызывает **assert** для проверки успешности операции **new**; вызывает **strcpy** для копирования исходной строки во вновь выделенную и **strcat** для конкатенации ее со строкой второго операнда; использует **delete**, чтобы освободить место, занятое исходной строкой объекта, и возвращает ***this** как **String &**, чтобы была возможность конкатенации операций +=.

Нужна ли нам вторая перегруженная операция, чтобы разрешить конкатенацию **String** и **char***? Нет. Конструктор преобразования **const char *** преобразует обычную строку в объект **String**, который после этого может участвовать в существующей операции конкатенации. C++ может выполнять такие преобразования только на один уровень глубины, чтобы удовлетворить требования к типам операндов. C++ может также производить неявное, определенное компилятором преобразование между встроенными типами перед тем, как он выполнит преобразование между встроенным типом и классом.

Совет по повышению эффективности 18.2

Перегрузка операции конкатенации **+=**, которая получает единственный аргумент типа **const char ***, работает более эффективно, чем использование неявного преобразования с последующей конкатенацией. Для неявных преобразований требуется меньше программного кода и они менее уязвимы для ошибок.

Строка программы

```
int operator!() const;// является ли String пустой?
```

объявляет перегруженную операцию отрицания. Эта операция обычно используется с объектами **String** для проверки содержимого строки. Например, когда компилятор встречает выражение **!string1**, он генерирует вызов функции

```
string1.operator!()
```

Эта функция просто возвращает результат проверки равенства нулю длины **(length)** строки.

Строки программы

```
int operator==(const String &) const;  // проверка s1 == s2
int operator!=(const String &) const;  // проверка s1 != s2
int operator<(const String &)  const;  // проверка s1 < s2
int operator>(const String &)  const;  // проверка s1 > s2
int operator>=(const String &) const;  // проверка s1 >= s2
int operator<=(const String &) const;  // проверка s1 <= s2
```

объявляют перегруженные операции равенства и отношений для класса **String**. Они все аналогичны, так что давайте обсудим только один пример — перегруженную операцию **>=**. Когда компилятор встречает выражение **string1> = string2**, он генерирует вызов функции

```
string1.operator>=(string2)
```

который возвращает **1 (true)**, если строка **string1** больше или равна **string2**. Каждая из этих операций использует **strcmp** для сравнения строк символов в объектах **String**.

Строка программы

```
char &operator[] (int);// возврат ссылки char
```

объявляет перегруженную операцию индексации. Когда компилятор встречает такое выражение, как **string1 [0]**, он генерирует вызов функции **string1.operator [] (0)**. Сначала функция **operator[]** вызывает **assert** для проверки диапазона индексации; если индекс выходит из диапазона, программа выведет сообщение об ошибке и аварийно завершит работу. Если индекс находится в допустимых пределах, **operator[]** возвращает как **char &** соответствующий символ объекта **String**; **char &** может использоваться как *lvalue* для изменения указанного символа в объекте **String**.

В строке программы

```
String &operator()(int, int); // возврат подстроки
```

объявляется *перегруженная операция вызова*. В классах строк перегрузка этой операции является обычной процедурой для выбора подстроки в объекте **String**. Два целых значения, использующиеся в качестве параметров функции, определяют начальное местоположение и длину подстроки, выбираемой из **String**. Если местоположение начала выходит из допустимых границ или длина подстроки имеет отрицательное значение, то генерируется сообщение об ошибке. В соответствии с соглашением, если длина подстроки равна 0, то подстрока выбирает все символы до конца объекта **String**.

Предположим, например, что **string1** является объектом **String**, содержащим строку символов **"AEIOU"**. Когда компилятор встречает выражение **string1(2, 2)**, он генерирует вызов функции **string1.operator( )(2, 2)**. При его выполнении создается временный объект **String**, содержащий строку **"IO"**, и возвращается ссылка на этот объект. Перегрузка функции-операции вызова **()** является сильным средством, поскольку функции могут получать произвольно длинный и сложный список параметров. Мы можем использовать эту особенность для многих интересных целей. Другим применением функции-операции вызова является альтернативная нотация индексации массива: вместо использования неуклюжих двойных квадратных скобок C для двумерных массивов типа **a [b] [c]** некоторые программисты предпочитают перегрузить функцию-операцию вызова, чтобы разрешить нотацию **a(b, c)**. Перегруженная функция-операция вызова может быть только не-статической функцией-элементом. Эта операция используется только тогда, когда «имя функции» является объектом класса **String**.

Строка программы

```
int getLength() const; // возврат длины строки
```

объявляет функцию, которая возвращает длину объекта **String**. Обратите внимание, что эта функция возвращает значение закрытых данных класса **String**.

Теперь читатель должен изучить программный код **main**, исследовать окно вывода и проверить каждый из вариантов применения перегруженных операций.

18.11. Перегрузка ++ и --

Все четыре типа операций приращения (инкремента) и уменьшения (декремента) — преинкремент и постинкремент, предекремент и постдекремент — могут быть перегружены. Мы увидим, как компилятор определяет, какая форма — префиксная или постфиксная — используется в операции приращения или уменьшения.

Чтобы перегрузить операции приращения, разрешив тем самым использование и преинкремента и постинкремента, каждая перегруженная функция-операция должна иметь различную запись, чтобы компилятор мог определить, какая из операций ++ подразумевается. Префиксные версии перегружаются точно так же, как перегружалась бы любая другая префиксная одноместная операция.

Предположим, что мы хотим прибавить 1 к значению дня, которым является объект **d1** класса **Date** (дата). Когда компилятор встречает выражение преинкремента

```
++d1
```

он генерирует вызов функции-элемента

```
d1.operator++()
```

Прототипом этой функции было бы

```
Date operator++();
```

Если преинкремент реализуется как функция, не являющаяся элементом класса, то, когда компилятор встречает выражение

```
++d1
```

он генерирует вызов функции

```
operator++(d1)
```

Прототип этой функции мог бы быть объявлен в классе **Date** как

```
friend Date operator++(Date &);
```

Перегрузка операции постинкремента представляет некоторую сложность, поскольку компилятор должен видеть разницу между сигнатурами перегруженных функций-операций преинкремента и постинкремента. Соглашение, которое было принято в C++, состоит в том, что, когда компилятор встречает выражение постинкремента

```
d1++
```

он будет генерировать вызов функции-элемента

```
d1.operator++(0)
```

Прототипом этой функции была бы запись

```
Date operator++(int)
```

Здесь 0 является просто значением-пустышкой, чтобы список аргументов **operator++**, использующийся для постинкремента, был отличим от списка аргументов **operator++** для преинкремента.

Если постинкремент реализуется как функция, не являющаяся элементом, то, когда компилятор встречает выражение

```
d1++
```

он генерирует вызов функции

```
operator++(d1, 0)
```

Прототипом этой функции было бы

```
friend Date operator++(Date &, int);
```

Повторим, что аргумент **0** типа **int** используется компилятором, чтобы список аргументов **operator++**, используемый для операции постинкремента, отличался бы от списка аргументов для операции преинкремента.

Все, что мы сказали в этом разделе по поводу перегрузки операций преинкремента и постинкремента, применимо и для перегрузки операций предекре-

мента и постдекремента. В следующем разделе мы исследуем класс **Date**, который объявляет перегруженные операции преинкремента и постинкремента.

18.12. Пример: класс Date

На рис. 18.6 показан класс **Date**. Класс использует перегруженные операции преинкремента и постинкремента, чтобы увеличить на 1 значение даты в объекте класса **Date**.

Класс содержит следующие функции-элементы: перегруженную операцию передачи в поток, конструктор по умолчанию, функцию **setDate**, перегруженную функцию-операцию преинкремента, перегруженную функцию-операцию постинкремента и перегруженную операцию присваивания со сложением (+=).

Программа-тестер в **main** создает объект-дату **d1**, который инициализируется значением January 1, 1990; объект-дату **d2**, который инициализируется значением December 27, 1992, и **d3**, который инициализируется значением недействительной даты. Конструктор **Date** вызывает **setDate** для проверки указанного месяца, дня и года. Если месяц недействителен, то его значением становится 1. Недействительное значение года заменяется на 1900. Недействительный день заменяется на 1.

```
// DATE1.H
// Определение класса Date
#ifndef DATE1_H
#define DATE1_H
#include <iostream.h>

class Date {
   friend ostream &operator<<(ostream &, const Date &);
public:
   Date(int m = 1, int d = 1, int y = 1900); // конструктор по
                                             // умолчанию
   void setDate(int, int, int);  // установить дату
   Date operator++();            // преприращение
   Date operator++(int);         // постприращение
   Date &operator+=(int); // добавить дни, модифицировать объект
private:
   int month;
   int day;
   int year;
   static int days[];      // массив дней в месяце
   void helpIncrement();   // сервисная функция
};

#endif
```

Рис. 18.6. Определение класса **Date** (часть 1 из 5)

Программа-тестер выводит каждый из созданных объектов **Date** с помощью перегруженной операции передачи в поток. Чтобы добавить 7 дней к **d2**, к нему применяется перегруженная операция +=. После этого, чтобы установить **d3** на February 28, 1992, вызывается функция **setDate**. Затем новый объект **Date**, **d4**, устанавливается на March 18, 1969. После этого **d4** увеличивает-

ся на 1 с помощью перегруженной операции приращения. Полученная дата выводится как до, так и после выполнения операции преинкремента для подтверждения того, что все сделано правильно. Наконец, следует приращение **d4** с использованием перегруженной операции постинкремента. Для проверки того, что все сделано правильно, полученная дата выводится как до, так и после выполнения операции постинкремента.

```
// DATE1.CPP
// Определения функций-элементов для класса Date
#include <iostream.h>
#include "date1.h"

// Инициализация статических элементов в области действия файла;
// одна копия на класс.
int Date::days[] = {0, 31, 28, 31, 30, 31, 30,
                    31, 31, 30, 31, 30, 31};

// конструктор Date
Date::Date(int m, int d, int y) { setDate(m, d, y); }

// Установить дату
void Date::setDate(int mm, int dd, int yy)
{
   month = (mm >= 1 && mm <= 12) ? mm : 1;
   year = (yy >= 1900 && yy <= 2100) ? yy : 1900;

   // проверка на изменение года
   if (month == 2 && year % 4 == 0 && (year % 400 == 0 ||
                                       year % 100 != 0))
      day = (dd >= 1 && dd <= 29) ? dd : 1;
   else
      day = (dd >= 1 && dd <= days[month]) ? dd : 1;
}

// Операция преинкремента перегружается как функция-элемент.

Date Date::operator++()
{
   helpIncrement();
   return *this;  // возврат значения, а не возврат ссылки
}

// Операция постинкремента перегружается как функция-элемент.
// Отметьте, что пустой целочисленный параметр не имеет имени.

Date Date::operator++(int)
{
   Date temp = *this;
   helpIncrement();

   // возврат временного объекта
   return temp; // возврат значения, а не ссылки
}
```

Рис. 18.6. Определения функций-элементов для класса **Date** (часть 2 из 5)

```
// Добавить к дате указанное число дней

Date &Date::operator+=(int additionalDays)
{
   for (int i = 1; i <= additionalDays; I++)
      helpIncrement();

   return *this;     // разрешить конкатенацию
}

// Функция для инкремента даты

void Date::helpIncrement()
{
   // проверка смены года
   if (month == 2 && day < 29 && year % 4 == 0 &&
       (year % 400 == 0 || year % 100 != 0))
      ++day;
   else if (day < Date::days[month])  // не последний день месяца
      ++day;
   else if (month < 12) {  // последний день месяца < December
      ++month;
      day = 1;
   }
   else {                  // December 31
      month = 1;
      day = 1;
      ++year;
   }
}

// Перегруженная операция вывода
ostream &operator<<(ostream &output, const Date &d)
{
   static char *monthName[13] = {"", "January",
      "February", "March", "April", "May", "June",
      "July", "August", "September", "October",
      "November" , "December" } ;

   output << monthName[d.month] << ' '
          << d.day << ", " << d.year;

   return output;   // разрешить конкатенацию
}
```

Рис. 18.6. Определения функций-элементов для класса **Date** (часть 3 из 5)

Перегрузка операции преинкремента является очевидной. Программа вызывает сервисную функцию **helpIncrement**, чтобы сделать фактическое приращение. Эта функция должна иметь дело с системой счисления дат, т.е. с правилами переноса, которые должны выполняться, когда мы пытаемся увеличить значение дня, уже достигшего максимального значения. В этом случае правила переноса требуют, чтобы было увеличено значение месяца. Если значение месяца уже равно 12, то значение года должно быть также увеличено.

```
// FIG18_6.CPP
// Тестер для класса Date
#include <iostream.h>
#include "date1.h"

main()
{
   Date d1, d2(12, 27, 1992), d3(0, 99, 8045);
   cout << "d1 is " << d1
        << "\nd2 is " << d2
        << "\nd3 is " << d3;
   cout << "\n\nd2 += 7 is " << (d2 += 7) << "\n\n";

   d3.setDate(2, 28, 1992);
   cout << "  d3 is " << d3;
   cout << "\n++d3 is " << ++d3 << "\n\n";

   Date d4(3, 18, 1969);

   cout << "Testing the preincrement operator:\n"
        << "  d4 is " << d4
        << "\n++d4 is " << ++d4
        << "\n  d4 is " << d4;

   cout << "\n\nTesting the postincrement operator:\n"
        << "  d4 is " << d4
        << "\nd4++ is " << d4++
        << "\n  d4 is " << d4 << endl;

   return 0;
}
```

Рис. 18.6. Тестер для класса **Date** (часть 4 из 5)

```
d1 is January 1, 1900
d2 is December 27, 1992
d3 is January 1, 1900

d2 += 7 is January 3, 1993

  d3 is February 28, 1992
++d3 is February 29, 1992

Testing the preincrement operator:
  d4 is March 18, 1969
++d4 is March 19, 1969
  d4 is March 19, 1969

Testing the postincrement operator:
  d4 is March 19, 1969
d4++ is March 19, 1969
  d4 is March 20, 1969
```

Рис. 18.6. Вывод программы-тестера для класса **Date** (часть 5 из 5)

Перегруженная операция преинкремента возвращает приращенную *копию* текущего объекта. Это происходит потому, что текущий объект, ***this**, возвращается как **Date**. Это фактически вызывает конструктор копии.

Процедура перегрузки операции постинкремента несколько сложнее. Чтобы имитировать эффект постинкремента, мы должны вернуть неувеличенную копию объекта **Date**. На входе в **operator++** мы записываем текущий объект (***this**) в **temp**. После этого мы вызываем **helpIncrement**, чтобы дать объекту приращение. Затем мы возвращаем немодифицированную копию объекта в **temp**.

Резюме

- Операция << в C++ может иметь различный смысл и используется в качестве операции передачи в поток и операции сдвига влево. Это пример *перегруженной операции*. Операция >> также перегружена, поскольку она используется как операция *извлечения из потока* и как операция правого сдвига.
- Вообще говоря, C++ дает программисту возможность перегружать большинство операций, чтобы их можно было применять к различным типам. Компилятор генерирует соответствующий программный код, исходя из контекста применения данной операции.
- Перегрузка операций способствует расширяемости C++
- Операции перегружаются посредством определения функции (с заголовком и телом). Ключевое слово **operator** с последующим символом перегружаемой операции становится именем функции.
- Чтобы использовать операцию с объектами класса, эта операция *должна* быть перегружена, но здесь есть два исключения. Операция присваивания (=) может использоваться с любым классом без перегрузки для выполнения поэлементного копирования по умолчанию. Операция взятия адреса (&) также может использоваться с объектами любого класса без перегрузки; эта операция просто возвращает адрес объекта в памяти.
- Смысл перегрузки операций — обеспечение той же лаконичности выражений для типов, определяемых пользователем, которую обеспечивает C++ для встроенных типов за счет богатого набора операций.
- Большая часть операций C++ может быть перегружена.
- Приоритет и ассоциативность операций не могут быть изменены посредством перегрузки.
- Для перегруженных операций нельзя использовать аргументы по умолчанию.
- Нельзя изменить число операндов, которое задано операцией; одноместная операция остается одноместной при перегрузке, а двухместная остается двухместной.
- Нельзя создавать обозначения для новых операций; только существующие операции могут быть перегружены.
- Способ работы операции с объектами встроенных типов не может быть изменен посредством перегрузки.

- При перегрузке (), [], -> или = перегружающая операцию функция должна объявляться как элемент класса.
- Функции-операции могут быть как функциями-элементами, так и функциями, не являющимися элементами.
- Когда функция-операция реализуется в качестве функции-элемента, левый (или единственный) операнд должен быть объектом (или ссылкой на объект), принадлежащим к классу этой операции.
- Если необходимо, чтобы левый операнд был объектом другого класса, то данная функция-операция должна применяться как функция, не являющаяся элементом класса.
- Функция-операция — элемент класса вызывается только тогда, когда левый операнд двухместной операции является некоторым объектом этого класса, или когда единственный операнд одноместной операции является объектом этого класса.
- Одна из причин, по которой для перегрузки операции можно выбрать функцию, не являющуюся элементом — требование коммутативности данной операции.
- Одноместная операция для класса может быть перегружена как не-статическая функция-элемент, не имеющая аргументов, или как функция, не являющаяся элементом и имеющая один аргумент. Этот аргумент должен быть либо объектом класса, либо ссылкой на объект.
- Двухместная операция может быть перегружена как не-статическая функция-элемент с одним аргументом или как функция, не являющаяся элементом и имеющая два аргумента (один из этих аргументов должен быть либо объектом класса, либо ссылкой на него).
- Операцию индексации массива [] не запрещено использовать и для других целей, например, для выбора элементов из разного рода контейнерных классов, таких, как связанные списки, строки, словари и т.д. Индексам также не обязательно быть только целыми, например, могут быть символами и строками.
- Конструктор копии инициализирует объект, используя другой объект того же класса. Конструктор копии вызывается также всякий раз, когда необходимо копирование объекта: при вызове по значению и при возврате объекта из вызванной функции.
- Компилятор не может автоматически определить, как выполнять преобразования между типами, определяемыми пользователем, и встроенными типами. Программист должен определить в явном виде, как проводить такие преобразования. Эти преобразования могут осуществляться с использованием конструктора преобразования, т.е. конструктора с одним аргументом, который просто превращает объект некоторого типа в объект данного класса.
- Операция преобразования (или приведения) может использоваться для преобразования объекта данного класса в объект другого класса или в объект встроенного типа. Такого рода операция преобразования должна быть не-статической функцией-элементом; этот вид операции преобразования не может быть функцией-другом.

- Конструктор преобразования — это конструктор с единственным аргументом, использующийся для преобразования аргумента в объект класса, которому принадлежит конструктор. Компилятор может вызывать такой конструктор неявным образом.
- Операция присваивания — наиболее часто перегружаемая операция. Обычно она используется для присваивания одного объекта другому объекту того же класса, но, посредством использования конструктора преобразования, она также может применяться в качестве операции присваивания объектов различных классов.
- Если перегруженная операция присваивания не определена, то присваивание разрешено, но, по умолчанию, присваивание будет просто поэлементным копированием каждого элемента данных. В некоторых случаях это приемлемо. Для объектов, которые содержат указатели на динамически выделенную память, поэлементное копирование даст в результате два различных объекта, указывающих на ту же самую динамически выделенную память. Вызов деструктора для любого из этих объектов очищает выделенную память. Если после этого другой объект обратится к этой области памяти, то результат такого обращения будет неопределенным.
- Чтобы перегрузить операции инкремента, разрешив тем самым использование и преинкремента и постинкремента, каждая перегруженная функция-операция должна иметь различную сигнатуру, чтобы компилятор мог определить, какая из операций ++ подразумевается. Префиксные версии перегружаются точно так же, как перегружаась бы любая другая префиксная одноместная операция. Обеспечение уникальной сигнатуры для функции-операции постинкремента достигается посредством второго аргумента, который должен быть целым **(int)**. На самом деле пользователь не передает значение этого специального аргумента целого типа. Это просто помогает компилятору различить префиксные и постфиксные версии операций инкремента и декремента.

Терминология

встроенный тип
класс **Array**
класс **Date**
класс **String**
ключевое слово **operator**
конкатенация вызовов функций
конструктор копии
конструктор преобразования
неперегружаемые операции
неявные преобразования типа
операции как функции
операция преобразования
перегружаемые операции
перегруженная дружественная функция-операция
перегруженная операция !=
перегруженная операция []
перегруженная операция <
перегруженная операция <=
перегруженная операция ==
перегруженная операция >
перегруженная операция >=
перегруженная операция присваивания (=)
Перегруженная функция-операция — элемент класса
перегрузка
перегрузка двухместной операции
перегрузка одноместной операции
перегрузка операций
перегрузка постфиксной операции
перегрузка префиксной операции
перегрузка функций

преобразование, определяемое пользователем
преобразования между встроенными типами и классами
преобразования между классами различного типа
явные преобразования типа (операция приведения)
разрешение неоднозначности преобразования
тип, определяемый пользователем

Распространенные ошибки программирования

18.1. Попытка создать новую операцию.

18.2. Попытка модифицировать способ работы операции с объектами встроенных типов.

18.3. Предположение, что перегрузка операций (таких как +) автоматически перегружает соответствующие операции присваивания (такие как +=). Операции могут быть перегружены только явным образом, а неявная перегрузка отсутствует.

18.4. Отсутствие проверки и обхода самоприсваивания при перегрузке операции присваивания класса, который содежит указатель на динамическую память.

18.5. Отсутствие перегруженной операции присваивания и конструктора копии класса, когда объекты этого класса содержат указатели на динамическую размещаемую память.

Хороший стиль программирования

18.1. Используйте перегрузку операции, когда это помогает сделать программу яснее, чем при выполнении тех же действий посредством явного вызова функции.

18.2. Избегайте избыточного или непоследовательного использования перегрузки операций, так как это может сделать программу трудно читаемой.

18.3. Перегружайте операцию для выполнения тех же действий с объектами класса, или близких по смыслу действий, которые выполняет эта операция со встроенными типами.

18.4 Перед тем как писать программу на C++, перегружающую операции, изучите руководство по вашему компилятору, чтобы учесть различные ограничения и требования, накладываемые на конкретные операции.

18.5. При перегрузке одноместных операций предпочтительнее делать функции-операции элементами класса, а не дружественными функциями, не являющимися элементами. Это более аккуратное решение. Нужно избегать дружественных функций и дружественных классов, если только это не является крайне необходимым. Использование друзей нарушает инкапсуляцию класса.

18.6. Деструктор, операция присваивания и конструктор копии класса обычно реализуются как единая группа.

Советы по повышению эффективности

18.1. Операцию можно перегрузить как функцию, не являющуюся элементом или дружественной функцией, но, если такой функции требуется доступ к закрытым или защищенным данным класса, ей придется обращаться к открытому интерфейсу этого класса (функциям вроде get или set). Такое опосредованное обращение к данным может привести к потере производительности.

18.2. Перегрузка операции конкатенации +=, которая получает единственный аргумент типа **const char ***, работает более эффективно, чем использование неявного преобразования с последующей конкатенацией. Для неявных преобразований требуется меньше программного кода и они менее уязвимы для ошибок.

Общие методические замечания

18.1. По меньшей мере один аргумент функции-операции должен быть объектом класса или должен ссылаться на объект класса. Это делает невозможным изменение способа воздействия операции на объекты встроенных типов.

18.2. Новые возможности ввода/вывода типов, определяемых пользователем, могут быть добавлены в C++ без модификации объявлений или элементов данных **private** в любом из классов ostream или **istream**. Это способствует расширяемости C++ как языка программирования, что является одним из наиболее привлекательных аспектов C++.

18.3. Имеется возможность избежать присваивания одного объекта класса другому. Это можно сделать, определив операцию присваивания как закрытый элемент класса.

Упражнения для самоконтроля

18.1. Заполните пропуски в следующих предложениях:

a) Предположим, что **a** и **b** — переменные целого типа и мы формируем сумму **a + b**. Теперь предположим, что **c** и **d** — переменные с плавающей точкой и мы формируем сумму **c + d**. Очевидно, что в данном случае две операции + используются для различных целей. Это является примером _____.

b) Ключевое слово _____ начинает определение перегруженной функции-операции.

c) Чтобы использовать операции на объектах класса, они должны быть перегружены, за исключением операций _____ и _____.

d) _____ и _____ операций не могут быть изменены перегрузкой.

18.2. Поясните неоднозначность операций << и >> в C++.

18.3. В каком контексте может использоваться имя **operator/** в C++?

18.4. (Верно/неверно) В C++ могут быть перегружены только уже существующие операции.

18.5. Как соотносится приоритет перегруженной операции в C++ с приоритетом первоначальной операции?

Ответы к упражнения для самоконтроля

18.1. a) перегрузки операций b) **operator** c) присваивание (=), адрес (&), d) Приоритет, ассоциативность

18.2. Операция >> является как операцией сдвига вправо, так и операцией извлечения из потока в зависимости от контекста. Операция << является как операцией сдвига влево, так и операцией передачи в поток.

18.3. Для перегрузки операции: это было бы именем функции, которая дала бы новую версию операции /.

18.4. Верно.

18.5. Идентичны.

Упражнения

18.6. Приведите как можно больше примеров неявной перегрузки операций в C. Приведите сколько сможете примеров неявной перегрузки операций в C++. Дайте мотивированный пример ситуации, в которой вы могли бы захотеть перегрузить операцию в C++ явным образом.

18.7. Операциями C++, которые не могут быть перегружены, являются _____, _____, _____, _____ и _____.

18.8. Для конкатенации строк требуются два операнда — две строки, которые должны быть соединены. В тексте мы показали, как применить перегруженную операцию конкатенации, которая присоединяет второй объект **String** в конец первого объекта **String**, изменяя тем самым первый объект **String**. В некоторых приложениях желательно получить объединенный объект **String** без изменений двух передаваемых аргументов **String**. Примените operator+ таким образом, чтобы были разрешены действия типа

```
string1 = string2 + string3;
```

18.9. (*Упражнение по оптимальной перегрузке операций*) Чтобы оценить осторожность, которая требуется при отборе операций для перегрузки, перечислите операции C++, которые являются перегружаемыми, и для каждой операции перечислите ее допустимые применения для каждого из нескольких классов, которые вы изучили в этом курсе. Мы предлагаем, чтобы вы попробовали это сделать для следующих классов:

a) Array
b) Stack
c) String

После выполнения этого упражнения прокомментируйте, какие операции имеют важное значение для широкого разнообразия классов. Какие операции имеют небольшую ценность в качестве перегруженных? Какие операции кажутся неоднозначными?

18.10. Теперь проведите процесс, описанный в предыдущей задаче, в обратном направлении. Перечислите все операции C++, которые являются перегружаемыми. Для каждой из них напишите, какое действие, по вашему мнению, может быть представлено данной операцией наилучшим образом. Если имеется несколько наилучших вариантов, внесите в список все эти действия.

18.11. (*Проект*) C++ — язык в процессе развития, и, кроме того, постоянно разрабатываются новые языки. Какие дополнительные операции вы могли бы рекомендовать добавить в C++, *или* в будущий язык, подобный C++, который поддерживал бы как процедурное, так и объектно-ориентированное программирование? Напишите аргументированное обоснование. Вы могли бы рассмотреть возможность отправки ваших предложений в Комитет ANSI C++.

18.12. Одним из хороших примеров применения перегруженной функции-операции вызова ( ) является возможность использовать более привычную форму записи двумерных массивов. Вместо нотации

```
chessBoard [row] [column]
```

для массива объектов перегрузите функцию-операцию вызова таким образом, чтобы можно было использовать альтернативную форму

```
chessBoard (row, column)
```

18.13. Перегрузите операцию индексации, чтобы вернуть указанный элемент связанного списка.

18.14. Перегрузите операцию индексации, чтобы вернуть наибольший элемент списка, второй *по величине* элемент, третий и т.д.

18.15. Рассмотрите класс **Complex**, показанный на рис. 18.7. Этот класс позволяет производить действия над комплексными числами, которые записываются в форме **realPart + imaginaryPart * *i***, где *i* имеет *значение* $\sqrt{-1}$.

а) Измените класс таким образом, чтобы разрешить ввод и вывод комплексных чисел посредством перегруженных операций >> и << (вы должны удалить из класса функцию печати).

b) Перегрузите операцию умножения таким образом, чтобы стало возможным умножение двух комплексных чисел по стандартным правилам алгебры.

c) Перегрузите операции == и != для сравнения комплексных чисел.

```
// COMPLEX1.H
// Определение класса Complex
#ifndef COMPLEX1_H
#define COMPLEX1_H

class Complex {
public:
```

```
   Complex(double = 0.0, double = 0.0);       // конструктор
   Complex operator+(const Complex &) const; // сложение
   Complex operator-(const Complex &) const; // вычитание
   Complex &operator=(const Complex &);       // присваивание
   void print() const;                        // вывод
private:
   double real;       // действительная часть
   double imaginary;  // мнимая часть
};

#endif
```

Рис. 18.7. Определение класса **Complex** (часть 1 из 4)

```
// COMPLEX1.CPP
// Определения функций-элементов для класса Complex
#include <iostream.h>
#include "complex1.h"

// Конструктор
Complex::Complex(double r, double i)
{
   real = r;
   imaginary = I;
}

// Перегруженная операция сложения
Complex Complex::operator+(const Complex &operand2) const
{
   Complex sum;
   sum.real = real + operand2.real;
   sum.imaginary = imaginary + operand2.imaginary;
   return sum;
}

// Перегруженная операция вычитания
Complex Complex::operator-(const Complex &operand2) const
{
   Complex diff;
   diff.real = real - operand2.real;
   diff.imaginary = imaginary - operand2.imaginary;
   return diff;
}

// Пререгруженная операция =
Complex& Complex::operator=(const Complex &right)
{
   real = right.real;
   imaginary = right.imaginary;
   return *this;   // разрешить конкатенацию
}

// Вывести объект Complex в виде: (a, b)
void Complex::print() const
   { cout << '(' << real << ", " << imaginary << ')'; }
```

Рис. 18.7. Определения функций-элементов для класса **Complex** (часть 2 из 4)

```
// FIG18_7.CPP
// Тестер для класса Complex
#include <iostream.h>
#include "complex1.h"

main()
{
   Complex x, y(4.3, 8.2), z(3.3, 1.1);

   cout << "x: ";
   x.print();
   cout << "\ny: ";
   y.print();
   cout << "\nz: ";
   z.print();

   x = y + z;
   cout << "\n\nx = y + z:\n";
   x.print();
   cout << " = ";
   y.print();
   cout << " + ";
   z.print();

   x = y - z;
   cout << "\n\nx = y - z:\n";
   x.print();
   cout << " = ";
   y.print();
   cout << " - ";
   z.print();
   cout << '\n';

   return 0;
}
```

Рис. 18.7. Тестер для класса **Complex** (часть 3 из 4)

```
x: (0, 0)
y: (4.3, 8.2)
z: (3.3, 1.1)

x = y + z:
(7.6, 9.3) = (4.3, 8.2) + (3.3, 1.1)

x = y - z;
(1, 7.1) = (4.3, 8.2) - (3.3, 1.1)
```

Рис. 18.7. Вывод программы-тестера для класса **Complex** (часть 4 из 4)

18.16. 32-разрядный компьютер может представлять целые числа в диапазоне приблизительно от -2 миллиардов до +2 миллиардов. Это ограничение достаточно редко создает сложности. Но имеется много приложений, в которых хотелось бы иметь возможность использовать намного более широкий диапазон целых чисел. В C++ встроена возможность создавать новые типы данных. Рассмотрите класс

HugeInt, приведенный на рис. 18.8. Тщательно изучите этот класс и после этого

a) Точно опишите, как он работает.

b) Какие ограничения имеет класс?

c) Измените класс таким образом, чтобы он мог обрабатывать произвольно большие целые числа. (Подсказка: используйте связанный список для представления **HugeInt**)

d) Перегрузите операцию умножения *.

e) Перегрузите операцию деления /.

f) Перегрузите все операции отношений и равенства.

```
// HUGEINT1.H
// Определение класса Hugeint
#ifndef HUGEINT1_H
#define HUGEINT1_H

#include <iostream.h>

class Hugeint {
   friend ostream &operator<<(ostream &, HugeInt &);
public:
   HugeInt(long = 0);              // конструктор преобразования
   HugeInt(const char *);          // конструктор преобразования
   HugeInt operator+(HugeInt &); // суммирование
private:
   short integer[30];
};

#endif

// HUGEINT1.CPP
// Определения Функций-элементов и друзей для класса HugeInt
#include <string.h>
#include "hugeint1.h"

// Конструктор преобразования
Hugeint::Hugeint(long val)
{
   for (int i = 0; i <= 29; i++)
      integer[i] = 0;    // инициализация массива обнулением

   for (i = 29; val != 0 && i >= 0; i--) {
      integer[i] = val % 10;
      val /= 10;
   }
}

HugeInt::HugeInt(const char *string)
{
   int i, j;

   for (i = 0; i <= 29; i++)
      integer[i] = 0;
```

Рис. 18.8. Пользовательский класс сверхбольших целых (часть 1 из 3)

```
   for (i = 30 - strlen(string), j= 0; i <= 29; i++, j++)
      integer[i] = string[j] - '0';
}

// Сложение
HugeInt HugeInt::operator+(HugeInt  &op2)
{
   HugeInt temp;
   int carry = 0;

   for (int i = 29; i >= 0; i--) {
      temp.integer[i] = integer[i] + op2.integer[i] + carry;

      if (temp.integer[i] > 9) {
         temp.integer[i] %= 10;
         carry = 1;
      }
      else
         carry = 0;
   }

   return temp;
}

ostream& operator<<(ostream &output, Hugeint &num)
{
   for (int i = 0; (num.integer[i] == 0) && (i <= 29); i++)
      ; // пропуск начальных нулей

   if (i == 30)
      output << 0;
   else
      for ( ; i <= 29; i++)
         output << num.integer [i];

   return output;
}

// FIG18_8.CPP
// Тестер для класса HugeInt
#include <iostream.h>
#include "hugeint1.h"

main()
{
   HugeInt n1(7654321), n2(7891234),
           n3("99999999999999999999999999999"),
           n4("1"), n5;

   cout << "n1 is " << n1 << "\nn2 is " << n2
        << "\nn3 is " << n3 << "\nn4 is " << n4
        << "\nn5 is " << n5 << "\n\n";

   n5 = n1 + n2;
   cout << n1 << " + " << n2 << " = " << n5 << "\n\n";
```

Рис. 18.8. Пользовательский класс сверхбольших целых (часть 2 из 3)

```
   cout << n3 << " + " << n4 << "\n= " << (n3 + n4) << "\n\n";

   n5 = n1 + 9;
   cout << n1 << " + " << 9 << " = " << n5 << "\n\n";

   n5 = n2 + "10000";
   cout << n2 << " + " << "10000" << " = " << n5 << "\n\n";

   return 0;
}
```

```
n1 is 7654321
n2 is 7891234
n3 is 99999999999999999999999999999
n4 is 1
n5 is 0

7654321 + 7891234 = 15545555

99999999999999999999999999999 + 1
= 100000000000000000000000000000

7654321 + 9 = 7654330
7891234 + 10000 = 7901234
```

Рис. 18.8. Пользовательский класс сверхбольших целых (часть 3 из 3)

18.17. Создайте класс **rationalNumber** (класс дробей) со следующими возможностями:

a) Создайте конструктор, который предотвращает появление 0 в знаменателе дроби, сокращает или упрощает дроби, которые имеют сокращаемую форму, и избегает появления отрицательных знаменателей.

b) Перегрузите операции сложения, вычитания, умножения и деления для этого класса.

c) Перегрузите операции отношения и равенства для этого класса.

18.18. Операция **sizeof** на объекте **String** возвращает только размер элементов данных объекта **String**, но не возвращает размер памяти **String**, отведенной под строку при вызове **new**. Перегрузите операцию **sizeof** таким образом, чтобы возвращать размер объекта **String**, включая динамически выделенную память.

18.19. Изучите функции библиотеки обработки строк C и реализуйте каждую из функций как часть класса **String**. Затем примените эти функции для обработки текста.

18.20. Разработайте класс **Polynomial**. Внутренним представлением **Polynomial** является связанный список членов полинома. Каждый член содержит коэффициент и степень. Член

$2x^4$

имеет коэффициент 2 и степень 4. Разработайте полный класс, содержащий соответствующие функции конструктора и деструктора,

а также функции *set* и *get*. Класс должен определять следующие перегруженные операции:

a) операцию сложения (+), чтобы складывать два объекта **Polynomial**;

b) операцию вычитания (-), чтобы вычитать один объект **Polynomial** из другого;

c) операцию присваивания, чтобы присваивать один объект **Polynomial** другому;

d) операцию умножения (*), чтобы умножать два объекта **Polynomial**;

e) операцию присваивания суммы (+=), операцию присваивания разности (-=) и операцию присваивания произведения (*=).

г л а в а

19

Наследование



Цели

- Научиться созданию новых классов, наследующих свойства уже существующих.
- Понять важность наследования для реализации принципа повторного использования программного кода.
- Изучить понятия базовых и производных классов.
- Изучить применение сложного наследования для создания производного класса на основе нескольких базовых.

Содержание

19.1. Введение
19.2. Базовые и производные классы
19.3. Защищенные элементы
19.4. Приведение указателей базового класса к указателям на производный класс
19.5. Использование функций-элементов
19.6. Переопределение элементов базового класса в производном классе
19.7. Открытые, защищенные и закрытые базовые классы
19.8. Непосредственные и косвенные базовые классы
19.9. Применение конструкторов и деструкторов в производных классах
19.10. Неявное преобразование объектов производного класса к базовому
19.11. Наследование в конструировании программного обеспечения
19.12. Композиция в сравнении с наследованием
19.13. Отношения «использует» и «знает»
19.14. Пример: Point, Circle, Cylinder
19.15. Сложное наследование

Резюме • *Распространенные ошибки программирования* • *Хороший стиль программирования* • *Совет по повышению эффективности* • *Общие методические замечания* • *Упражнения для самоконтроля* • *Ответы на упражнения для самоконтроля* • *Упражнения*

19.1. Введение

В этой и следующей главах мы обсудим две самые существенные особенности, которые характеризуют объектно-ориентированное программирование — это *наследование* и *полиморфизм*. Наследование представляет собой механизм повторного использования программного обеспечения, в соответствии с которым новые классы создаются на основе существующих. Эти классы наследуют свойства и поведение базовых классов и приобретают дополнительно новые, необходимые для новых классов, качества. Возможность повторного использования программного обеспечения сберегает время, затраченное на его разра-

ботку, способствует повторному использованию апробированного и отлаженного высококачественного программного обеспечения и уменьшает число эксплуатационных проблем, когда система начинает функционировать. Это сулит богатые возможности. Полиморфизм позволяет нам писать программы в общем виде для обработки большого разнообразия существующих и определяемых в дальнейшем логически связанных классов. Наследование и полиморфизм представляют собой эффективные методики для разработки сложных программных систем.

При создании нового класса вместо того, чтобы писать совершенно новые элементы данных и функции-элементы, программист может просто указать, что новый класс должен *наследовать* элементы ранее определенного *базового* класса. Этот новый класс называется *производным классом*. Каждый производный класс, в свою очередь, может быть базовым для каких-то будущих производных классов. При *простом наследовании* производный класс получается на основе только одного базового класса. При *сложном наследовании* производный класс наследует свойства от многих (возможно, логически не связанных) классов. Новый класс, как правило, вводит свои элементы данных и функции, поэтому производный класс, вообще говоря, «больше» исходного базового класса. Новый производный класс имеет больше специфических свойств в сравнении с исходным базовым и представляет меньшую группу объектов. В случае простого наследования производный класс в своей основе остается по существу таким же, как базовый класс. Настоящая сила наследования определяется возможностью добавлять, замещать и уточнять наследуемые от базовых классов свойства.

Каждый объект производного класса является также объектом базового класса, из которого получен этот производный класс. Однако противоположное неверно — объекты базового класса не являются объектами производных классов базового объекта. Мы воспользуемся этим отношением — «объект производного класса является объектом базового класса», чтобы выполнять некоторые интересные преобразования. Например, мы можем перечислить много разных объектов, связанных отношением наследования, в связанном списке объектов базового класса. Это позволит обрабатывать разные объекты одним общим образом. Как мы в дальнейшем увидим в этой и следующей главах, это является одним из важнейших методов объектно-ориентированного программирования.

Мы введем в этой главе новое средство для управления доступом к элементу — речь идет о защищенном доступе. Производные и дружественные им классы имеют право доступа к защищенным элементам базового класса; у остальных функций такого права нет.

Опыт построения программных систем показывает, что большие части программы имеют дело с тесно связанными специальными случаями. В таких системах трудно увидеть «целую картину», потому что проектировщик и программист заняты своими частностями. Объектно-ориентированное программирование предоставляет некоторые средства для того, чтобы «за деревьями видеть лес». Один такой процесс, позволяющий «видеть лес», называют иногда *абстракцией*.

Если программа перегружена тесно связанными специальными случаями, то обычно бросаются в глаза переключатели **switch**, которые распознают эти специальные случаи и обеспечивают обрабатывающую логику, которая занимается каждым случаем по отдельности. В главе 20 мы покажем, как, исполь-

зуя наследование и полиморфизм, более просто реализовать логику, содержащую операторы **switch**.

Мы делаем различие между *отношениями «является»* и *«имеет»*. Первое относится к наследованию. В отношении «является» объект типа производного класса можно обрабатывать как объект типа базового класса. Отношение «имеет» представляет композицию (см. рис. 17.4). В отношении «имеет» объект класса содержит один или несколько объектов других классов в качестве элементов.

Производный класс не может иметь доступа к закрытым элементам своего базового класса; такой доступ нарушал бы инкапсуляцию последнего. Однако производный класс может иметь доступ к открытым и защищенным элементам базового класса. Элементы базового класса, доступ к которым не разрешен для производного класса через отношение наследования, должны быть объявлены в базовом классе как закрытые. Производный класс может иметь доступ к закрытым элементам базового класса только через специальные функции доступа, предоставляемые открытым интерфейсом базового класса.

Одна проблема с наследованием заключается в том, что производный класс может наследовать открытые функции-элементы, которые не нужно или не следовало бы иметь в «потомстве» явно. Если какой-нибудь элемент базового класса не подходит для производного класса, этот элемент может быть переопределен в самом производном классе с соответствующей реализацией.

Наиболее замечательна сама идея, что новые классы могут наследовать от существующих *библиотек классов*. Организации разрабатывают свои собственные библиотеки классов, а могут использовать другие библиотеки, доступные во всем мире. Есть точка зрения, что когда-нибудь программное обеспечение будут создавать из стандартизованных повторно используемых компонентов точно так же, как сегодня часто собирают аппаратуру. Это поможет удовлетворить требования по разработке еще более мощного программного обеспечения, которое нам понадобится в будущем.

19.2. Базовые и производные классы

Часто объект одного класса «является» также и объектом другого класса. Прямоугольник — **rectangle**, конечно же, *является* четырехугольником — **quadrilateral** (как являются и квадрат, и параллелограм, и трапеция). Таким образом, о классе **Rectangle** можно сказать, что он наследует классу **Quadrilateral**. В этом смысле класс **Quadrilateral** называется *базовым*, а класс **Rectangle** называется *производным*. Прямоугольник *является* специальным типом четырехугольника, но нельзя исходя из этого утверждать, что четырехугольник *является* прямоугольником. Рис. 19.1 показывает несколько простых примеров отношения наследования.

Базовый класс	Производные классы
Студент	Живущий в городе
	Живущий при колледже
Фигура	Круг
	Треугольник
	Прямоугольник

Базовый класс	Производные классы
Ссуда	Ссуда на автомобиль
	Ссуда на ремонт дома
	Ссуда по закладной
Сотрудник	Член факультета
	Вспомогательный персонал
Счет	Текущий счет
	Сберегательный счет

Рис. 19.1. Некоторые простые примеры наследования

Другие языки объектно-ориентированного программирования, такие, как Smalltalk, используют другую терминологию: в отношении наследования базовый класс называется *надклассом*, а производный класс — *подклассом*. Поскольку наследование обычно образует производные классы «больше» базовых, термины «надкласс» и «подкласс», по-видимому, не очень удачны; мы будем избегать их применения.

Наследование приводит к древовидным иерархическим структурам. Базовый класс находится в иерархическом отношении со своими производными классами. Один класс, конечно, может существовать сам по себе, но когда класс включается в механизм наследования, он становится либо базовым классом, который определяет свойства и поведение других классов, либо производным классом, который наследует чьи-то свойства и поведение.

Построим один простой пример иерархии наследования (рис. 19.2). В типичном университетском сообществе тысячи людей, которые являются его членами (**CommunityMember**). Эти люди — сотрудники (**Employee**) и студенты (**Student**). Сотрудники относятся либо к профессорско-преподавательскому составу — сотрудники факультетов (**Faculty**), — либо к вспомогательному персоналу (**Staff**). Факультетские сотрудники являются либо администраторами (**Administrator**), либо преподавателями (**Teacher**). Это дает иерархию наследования, показанную на рис. 19.2.

Другой важный пример иерархии наследования дает понятие **формы** — частично иерархия этого класса показана на рис. 19.3. Как правило, студенты, впервые изучающие объектно-ориентированное програмирование, замечают, что в реальном мире существует огромное число примеров иерархий. Именно то обстоятельство, что они никогда не задумывались о такой категоризации мира, приводит к необходимости перестройки их мышления.

Для того, чтобы указать, что класс **CommissionWorker** получается из класса **Employee**, класс **CommissionWorker** обычно определяется следующим образом:

```
class CommissionWorker   : public Employee  {
  ...
};
```

Это пример *открытого наследования*. Вероятно, читателю часто придется им пользоваться. Мы также будем говорить о *защищенном* и *закрытом наследовании*. В случае открытого наследования открытые и защищенные элемен-

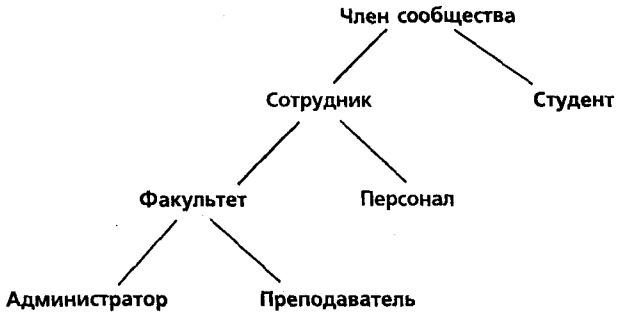


Рис. 19.2. Иерархия наследования для членов университетского сообщества



Рис. 19.3. Часть иерархии наследования для класса **Shape** (Форма)

ты базового класса наследуются в качестве открытых и защищенных элементов производным классом.

Можно рассматривать объекты базового класса и объекты производного класса аналогичным образом. Общность выражается в атрибутах и поведении базового класса. Объекты *любых* открыто-производных классов общего базового класса можно рассматривать как объекты *этого* базового класса. Мы рассмотрим много примеров, в которых мы используем достоинства этого отношения, заключающегося в легкости программирования, недостижимой в не-объектных языках, каким является C.

19.3. Защищенные элементы

Все функции программы имеют доступ к открытым элементам базового класса. Доступ к закрытым элементам базового класса имеют только элементы-функции самого класса и его «друзья».

Защищенный доступ представляет собой промежуточный уровень защиты между закрытым и открытым доступом. Защищенные элементы базового класса могут быть доступны только для элементов и друзей самого класса и для элементов и друзей производных классов.

Элементы производного класса могут вызывать открытые и защищенные элементы базового класса просто по имени элемента. При этом не обязательно использовать операцию разрешения области действия — по умолчанию имеется в виду текущий объект.

19.4. Приведение указателей базового класса к указателям на производный класс

Объект производного класса с открытым наследованием можно обрабатывать как объект соответствующего базового класса. Это позволяет выполнять некоторые интересные манипуляции. Например, несмотря на то, что объекты целого ряда классов, полученных из некоторого базового класса, могут совершенно отличаться друг от друга, мы все же можем создать из них связанный список, — если мы будем рассматривать их как объекты базового класса. Однако обратное неверно: объекты базового класса не являются автоматически объектами производного.

Распространенная ошибка программирования 19.1

Использование объекта базового класса как объекта производного класса.

Однако программист может применить явное приведение типа, чтобы преобразовать указатель базового класса в указатель производного. Но эту операцию надо выполнять аккуратно — если такой указатель должен быть разыменован, то нужно быть уверенным, что он указывает на объект производного класса.

Распространенная ошибка программирования 19.2

Использование элементов производного класса, не определенных в базовом классе, после преобразования указателя объекта базового класса к указателю на объект производного класса.

Наш первый пример показан на рис. 19.4, части 1–5. Части 1 и 2 показывают определение класса **Point** и определения его функций-элементов. Части 3 и 4 показывают определение класса **Circle** и определения функций-элементов **Circle**. Часть 5 представляет программу-тестер, в которой иллюстрируется присваивание указателя производного класса указателям базового класса и приведение указателей базового класса к указателям производного класса.

Сначала рассмотрим определение класса **Point** (рис. 19.4, часть 1). Открытый интерфейс для **Point** содержит функции-элементы **setPoint**, **getX** и **getY**. Элементы-данные **x** и **y** класса **Point** указаны как защищенные — **protected**. Это не позволяет пользователям объектов класса **Point** иметь прямой доступ к данным, но разрешает его производным классам непосредственно обращаться к унаследованным элементам данных. Если бы эти данные были определены как закрытые — **private**, то для доступа к данным нужно было бы вызывать открытые функции-элементы класса **Point**. Заметьте, что функция перегруженной операции передачи в поток объекта **Point** (рис. 19.4, часть 2) может прямо ссылаться на переменные **x** и **y** несмотря на то, что они — защищенные элементы класса **Point**. Заметьте также, что необходимо ссылаться на **x** и **y** через объект (то есть использовать запись **p.x** и **p.y**,) так как функция перегруженной операции вставки в поток является не функцией-элементом класса **Circle**, а дружественной ему функцией.

```
// POINT .H
// Определение класса Point
#ifndef POINT_H
#define POINT_H

class Point {
   friend ostream &operator<<(ostream &, const Point &);
public:
   Point (float = 0, float = 0);       // конструктор по умолчанию
   void setPoint (float, float);       // установка координат
   float getX() const { return x; }    // получение координаты x
   float getY() const { return y; }    // получение координаты y
protected:                  // доступна производному классу
   float x, y;              // x и y - координаты точки
};

#endif
```

Рис. 19.4. Определение класса **Point** (часть 1 из 5)

```
// POINT.CPP
// Функции-элементы класса Point
#include <iostream.h>
#include "point.h"

// Constructor for class Point
Point::Point(float a, flqat b)
{
   x = a;
   y = b;
}

// Set x and y coordinates of Point
void Point::setPoint(float a, float b)
{
   x = a;
   y = b;
}

// Вывод Point (перегруженная операция передачи в поток)
ostream &operator<<(ostream &output, const Point &p)
{
   output << '[' << p.x << ", " << p.y << ']';

   return output;   // разрешает конкатенацию вызовов
}
```

Рис. 19.4. Определения функций-элементов для класса **Point** (часть 2 из 5)

Класс **Circle** (Рис. 19.4, часть 3) выводится из класса **Point** по типу открытого наследования. Это указано в первой строке определения класса:

```
class Circle : public Point {  // Circle наследует от Point
```

Двоеточие (:) в заголовке определения класса указывает на наследование. Ключевое слово **public** определяет тип наследования (см. раздел 19.7). Все элементы класса **Point** наследуются классом **Circle**. Это означает, что открытый интерфейс класса **Circle** включает в себя открытые функции-элементы класса **Point** и, кроме того, функции-элементы **area, setRadius** и **getRadius.**

```
// CIRCLE.H
// Определение класса Circle
#ifndef CIRCLE_H
#define CIRCLE_H

#include <iostream.h>
#include <iomanip.h>
#include "point.h"

class Circle : public Point {  // Circle наследует от Point
   friend ostream &operator<<(ostream &, const Circle &);
public:
   // конструктор по умолчанию
   Circle(float r = 0.0, float x = 0, float y = 0);

   void setRadius(float);       // установка радиуса
   float getRadius() const;     // возврат радиуса
   float area() const;          // вычисление площади
protected:
   float radius;
};

#endif
```

Рис. 19.4. Определение класса **Circle** (часть 3 из 5)

```
// CIRCLE.CPP
// Определения функций-элементов Circle
#include "circle.h"

// Конструктор Circle вызывает конструктор Point
// с инициализатором элементов, затем инициализирует радиус.
Circle::Circle(float r, float a, float b)
   : Point(a, b)        // вызов базового конструктора
{ radius = r; }

// Установить радиус Circle
void Circle::setRadius(float r) { radius = r; }

// Получить радиус Circle
float Circle::getRadius() const { return radius; }

// Вычислить площадь Circle
float Circle::area() const
{ return 3.14159 * radius * radius; }

// Вывести Circle в форме:
// Center = [x, y]; Radius = #.##
ostream &operator<<(ostream &output, const Circle &c)
{
   output << "Center = [" << c.x << ", " << c.y
          << "]; Radius = " << setiosflags(ios::showpoint)
          << setprecision(2) << c.radius;

   return output;   // разрешает конкатенацию вызовов
}
```

Рис. 19.4. Определения функций-элементов для класса **Circle** (часть 4 из 5)

```
// FIG19_4.CPP
// Приведение указателей базового класса к производному

#include <iostream.h>
#include <iomanip.h>
#include "point.h"
#include "circle.h"

main ()
{
   Point *pointPtr, p(3.5, 5.3);
   Circle *circlePtr, c(2.7, 1.2, 8.9);

   cout << "Point p: " << p << "\nCircle c: " << c << endl;

   // Рассматривает Circle как Circle
   // (с промежуточным приведением)
   pointPtr = &c;      // присвоить Circle указателю pointPtr
   circlePtr = (Circle *) pointPtr;  // привести базовый
                                     // к производному
   cout << "\nArea of c (via circlePtr): "
        << circlePtr->area() << endl;

   // ОПАСНО: Рассматривает Point как Circle
   pointPtr = &p;    // присвоить адрес Point указателю pointPtr
   circlePtr = (Circle *) pointPtr;  // привести базовый
                                     // к производному
   cout << "\nRadius of object circlePtr points to: "
        << circlePtr->getRadius() << endl;

   return 0;
}
```

```
Point p: [3.5, 5.3]
Circle c: Center = [1.2, 8.9]; Radius << 2.70

Area of c (via circlePtr): 22.90

Radius of object circlePtr points to: 4/02e-38
```

Рис. 19.4. Приведение указателей базового класса к указателям производного (часть 5 из 5)

Конструктор **Circle** (см. рис. 19.4, часть 4) должен вызывать конструктор **Point**, чтобы инициализировать данные базового класса объекта **Circle**. Для этого используется синтаксис инициализатора элемента, описанный в главе 17, следующим образом:

```
Circle::Circle(float r, float a, float b)
   : Point(a, b)        // вызов базового конструктора
```

Вторая строка заголовка функции конструктора вызывает конструктор **Point** по имени. Значения параметров **a** и **b** передаются конструктором **Circle** конструктору **Point**, чтобы непосредственно инициализировать элементы **x** и **y** базового класса. Заметьте, что перегруженная функция операции передачи в поток **Circle** может непосредственно обращаться к элементам **x** и **y**, потому что они являются защищенными элементами класса **Point**, а функция дружест-

венна производному от него классу. Заметьте также, что необходимо ссылаться на элементы **x** и **y** через объект (**c.x** и **c.y**), поскольку функция перегруженной операции помещения в поток не является элементом класса **Circle**. Она только дружественна ему.

В приведенной программе-тестере (рис. 19.4, часть 5) создается указатель **pointPtr** на объект **Point** и объект **p** класса **Point**, затем создается указатель **circlePtr** на объект **Circle** и экземпляр объекта **c** класса **Circle**. Объекты классов **Point** и **Circle** выводятся с помощью перегруженных операций передачи в поток, чтобы показать правильность инициализации. Затем выполняется присваивание указателя производного класса (адрес объекта **c**) указателю **pointPtr** базового класса и приведение указателя **pointPtr** обратно к указателю на класс **Circle***. Результат операции приведения присваивается указателю **circlePtr**. Площадь объекта **c** класса **Circle** выводится через указатель **circlePtr**. Это приводит к правильному значению площади, так как указатели все время ссылаются на объекты производного класса. Присваивание указателя производного класса указателю базового всегда допустимо, так как объект производного класса *является* объектом базового класса. Указатель базового класса позволяет «видеть» только ту часть производного класса, которая унаследована от базового. Компилятор сам выполняет неявное преобразование указателя производного класса в указатель базового класса. Указатель базового класса нельзя прямо присвоить указателю производного класса, так как такое присваивание потенциально опасно — предполагается, что указатели производного класса указывают на объекты производного класса. В этом случае компилятор не выполняет неявного преобразования. Использование явного приведения показывает компилятору, что программист осознает опасность подобного преобразования указателя и учитывает все возможные его последствия.

Затем программа демонстрирует присваивание адреса объекта **p** класса **Point** указателю базового класса **pointPtr** и приведение **pointPtr** к типу **Circle***. Результат операции приведения присваивается указателю **circlePtr**. Радиус объекта, указанного **circlePtr** (объекта **p** класса **Point**) выводится через **circlePtr**. Это приводит к неправильному результату, так как все указатели ссылаются на объект класса **Point**. Объект класса **Point** не имеет элемента **radius**. Поэтому программа выводит любое значение, которое находится в памяти в том месте, где, как считает указатель **circlePtr**, находится элемент **radius**.

19.5. Применение функций-элементов

Если производный класс образуется из базового класса, функциям-элементам производного класса может потребоваться доступ к определенным элементам базового класса.

Общее методическое замечание 19.1

Производный класс не может иметь прямого доступа к закрытым элементам своего базового класса.

Это критический аспект технологии программного обеспечения в C++. Если бы производный класс мог иметь доступ к закрытым элементам базового класса, то это нарушало бы инкапсуляцию последнего. Возможность прятать закрытые элементы дает огромные преимущества в отношении тестирования,

отладки и корректной модификации систем. Если бы производный класс мог иметь доступ к закрытым элементам базового класса, то оказалось бы возможным, что классы, полученные из этого производного класса, имели бы также доступ и к этим данным и т.д. Это открыло бы доступ к тем данным, которые считаются закрытыми, и, таким образом, преимущество инкапсуляции было бы потеряно во всей иерархии классов.

19.6. Переопределение элементов базового класса в производном классе

Производный класс может переопределить функцию-элемент базового класса. Когда в производном классе употребляется имя этой функции, автоматически выбирается функция производного класса. Для доступа из производного класса к функции базового класса можно использовать операцию разрешения области действия.

Распространенная ошибка программирования 19.3

Когда функция-элемент базового класса переопределяется в производном классе, функция производного класса обычно вызывает одноименную функцию базового и выполняет некоторую дополнительную работу. В случае, когда для вызова функции базового класса не используется операция разрешения области действия, возникает бесконечная рекурсия, так как функция-элемент производного класса фактически вызывает саму себя.

Хороший стиль программирования 19.1

Когда в производном классе наследуются ненужные свойства, маскируйте их, переопределяя функции.

Общее методическое замечание 19.1

Переопределение функции-элемента в производном классе не требует одинаковой сигнатуры с функцией базового класса.

Рассмотрим упрощенный класс **Employee**. Он хранит имена и фамилии сотрудников — **firstName** и **lastName**. Эта информация является общей для всех сотрудников, в том числе и сотрудников, информация о которых содержится в производных от **Employee** классах. Из базового класса **Employee** построим теперь классы сотрудников **HourlyWorker**, **PieceWorker**, **Boss** и **CommissionWorker**. Сотрудники-почасовики **HourlyWorker** получают почасовую заработную плату и в случае, когда их переработка в неделю превышает 40 часов, эти дополнительные часы оплачиваются по расценкам, в полтора раза превышающим обычные. Класс сотрудников со сдельной оплатой **PieceWorker** получает фиксированную оплату за единицу выполненной работы — для простоты предположим, что каждый из них выполняет только один вид работы, так что закрытые элементы-данные будут количеством произведенных единиц и величиной оплаты за единицу. Класс сотрудников-начальников **Boss** получает фиксированную недельную ставку. Класс сотрудников по продаже **CommissionWorker** получает небольшую фиксированную недельную плату и дополнительно фиксированный

процент с проданного за неделю товара. Для простоты рассмотрим только два класса сотрудников — **Employee** и **HourlyWorker**.

Программный код для этого примера приведен на рис. 19.5, части 1–5. Части 1 и 2 показывают определение класса **Employee** и определения его элементов-функций. Части 3 и 4 — определение класса **HourlyWorker** и определения его функций-элементов. Часть 5 показывает программу-тестер иерархии наследования **Employee / HourlyWorker**, который просто создает экземпляр объекта класса **HourlyWorker**, инициализирует его и вызывает функцию-элемент **print** класса **HourlyWorker** для вывода данных об объекте.

```
// EMPLOY.H
// Определение класса Employee
#ifndef EMPLOY_H
#define EMPLOY_H

class Employee {
public:
   Employee(const char*, const char*);  // конструктор
   void print() const;    // вывод имени и фамилии
   ~Employee ();          // деструктор
private:
   char *firstName;      // динамически выделяемая строка
   char *lastName;       // динамически выделяемая строка
};

#endif
```

Рис. 19.5. Определение класса **Employee** (часть 1 из 5)

Определение класса **Employee** (рис. 19.5, часть 1) состоит из двух закрытых элементов-данных типа **char *** — **firstName** и **lastName** — и трех функций-элементов — конструктора, деструктора и функции **print**. Конструктор (рис. 19.5, часть 2) получает две строки и выделяет динамически массивы символов для хранения строк. Заметьте, что макрокоманда **assert** (рассмотренная в главе 14) используется здесь для того, чтобы определить, была ли выделена память для размещения **firstName** и **lastName**. В противном случае программа завершается сообщением об ошибке, в котором указаны проверяемоеое условие, номер строки, где возникло условие, и файл, в котором производится проверка. Поскольку данные класса **Employee** закрытые, доступ к ним возможен только через функцию-элемент **print**, которая просто выводит на печать имя и фамилию служащих. Деструктор возвращает системе динамически выделенную область памяти.

Класс **HourlyWorker** (рис. 19.5, часть 3) выводится из класса **Employee** с открытым наследованием. Это, как и в предыдущем примере, указывается в первой строке определения класса:

```
class HourlyWorker : public Employee
```

Открытый интерфейс для класса **HourlyWorker** включает функцию **print** класса **Employee** и функции-элементы **getPay** и **print** класса **HourlyWorker**. Обратите внимание, класс **HourlyWorker** определяет свою собственную функцию **print**. Таким образом, класс имеет доступ к двум функциям **print**. Кроме того, класс **HourlyWorker** содержит закрытые элементы-данные **wage** и **hours** для вычисления еженедельной заработной платы сотрудников.

```
// EMPLOY.CPP
// Member function definitions for class Employee
#include <string.h>
#include <iostream.h>
#include <assert.h>
#include "employ.h"

// Конструктор динамически выделяет пространство для
// имени и фамилии и вызывает strcpy для копирования
// имени и фамилии в объект.
Employee::Employee(const char *first, const char *last)
{
   firstName = new char[ strlen(first) + 1 ];
   assert(firstName != 0); // прерывание, если память не выделена
   strcpy(firstName, first);

   lastName = new char[ strlen (last) + 1 ];
   assert (lastName != 0); // прерывание, если память не выделена
   strcpy (lastName, last);
}

// Вывод имени сотрудника
void Employee::print() const
{ cout << firstName << ' ' << lastName; }

// Деструктор освобождает выделенную память
Employee::~Employee ()
{
   delete [] firstName;   // освободить динамическую память
   delete [] lastName;    // освободить динамическую память
}
```

Рис. 19.5. Определения функций-элементов класса **Employee** (часть 2 из 5)

```
// HOURLY.H
// Определение класса HourlyWorker
#ifndef HOURLY_H
#define HOURLY_H

#include "employ.h"

class HourlyWorker : public Employee {
public:
   HourlyWorker (const char*, const char*, float, float);
   float getPay() const; // вычисление и возврат значения
                         // зарплаты
   void print () const;  // переопределенная print
private:
   float wage;           // почасовая ставка
   float hours;          // рабочие часы за неделю
};

#endif
```

Рис. 19.5. Определение класса **HourlyWorker** (часть 3 из 5)

```
// HOURLY_B.CPP
// Определения функций-элементов HourlyWorker
#include <iostream.h>
#include <iomanip.h>
#include "hourly.h"

// Конструктор для HourlyWorker
HourlyWorker::HourlyWorker(const char *first, const char *last,
                           float initHours, float initWage)
   : Employee (first, last)  // вызов базового конструктора
{
   hours = initHours;
   wage = initWage;
}

// Получить зарплату HourlyWorker
float HourlyWorker::getPay() const { return wage * hours; }

// Напечатать имя и плату для HourlyWorker
void HourlyWorker::print() const
{
   cout << "HourlyWorker::print()\n\n";

   Employee::print();   // вызов базовой print

   cout << " is an hourly worker with pay of"
        << " $" << setiosflags(ios::showpoint)
        << setprecision(2) << getPay() << endl;
}
```

Рис. 19.5. Определения функций-элементов класса **HourlyWorker** (часть 4 из 5)

```
// FIG19_5.CPP
// Переопределение функций-элементов в производном классе
#include <iostream.h>
#include "hourly.h"

main ()
{
   HourlyWorker h("Bob", "Smith", 40.0, 7.50);
   h.print();
   return 0;
}
```

```
HourlyWorker::print()

Bob Smith is an hourly worker with pay of $300.00
```

Рис. 19.5. Переопределение функции-элемента базового класса в производном классе (часть 5 из 5)

Конструктор **HourlyWorker** (рис. 19.5, часть 4) использует синтаксис инициализатора элемента для передачи строк **first** и **last** конструктору класса **Employee**, чтобы инициализировать элементы базового класса, затем инициализирует элементы **wage** и **hours**. Функция-элемент **getPay** вычисляет заработную плату для **HourlyWorker**.

Функция-элемент **print** класса **HourlyWorker** представляет собой пример переопределения функции-элемента базового класса в производном классе. Часто функции-элементы базового класса переопределяются в производном классе для того, чтобы придать последнему большие функциональные возможности. Переопределяемые функции иногда вызывают базовую функцию для выполнения части своей задачи. В этом примере функция **print** производного класса вызывает функцию **print** базового для вывода на печать имени сотрудника (эта функция **print** базового класса является единственной функцией с доступом к закрытым данным базового класса). Функция **print** производного класса, кроме того, выводит на печать и заработную плату сотрудника. Обратите внимание, как вызывается функция **print** базового класса:

```
Employee::print();
```

Поскольку функции базового и производного классов имеют одинаковые имена, функции базового класса должно предшествовать имя этого класса с операцией разрешения области действия. В противном случае будет вызвана функция производного класса, что приведет к бесконечной рекурсии (функция **print** класса **HourlyWorker** вызывала бы саму себя).

19.7. Открытые, защищенные и закрытые базовые классы

При выводе из базового класса производный класс может наследовать открытый, защищенный *либо* закрытый базовый класс. Защищенный и закрытый тип наследования редко используются и применять их следует с большими предосторожностями. В этой книге мы используем в примерах только открытый тип наследования.

При выводе класса из открытого базового класса открытые элементы базового класса становятся открытыми элементами производного, а защищенные элементы базового класса становятся защищенными элементами производного класса. Из производного класса к закрытым элементам базового класса никогда нет доступа.

При выводе класса из защищенного базового класса открытые и защищенные элементы базового класса становятся защищенными элементами производного.

При выводе класса из закрытого базового класса открытые и защищенные элементы базового класса становятся закрытыми элементами производного.

19.8. Непосредственные и косвенные базовые классы

Базовый класс может быть или *непосредственным базовым классом* производного класса, или его *косвенным базовым классом*. Непосредственный базовый класс явно указывается в заголовке производного класса при его описании. *Косвенный базовый класс* не указывается явно в заголовке — точнее говоря, косвенный базовый класс наследуется с некоторого более высокого уровня иерархии классов.

19.9. Применение конструкторов и деструкторов в производном классе

Когда создается экземпляр объекта производного класса, то, поскольку последний наследует элементы своего базового класса, необходимо вызывать конструктор базового класса для инициализации базовых элементов объекта (производного класса). Конструктор производного класса может вызвать конструктор базового класса либо неявно, либо можно предусмотреть в конструкторе производного класса *инициализатор базового класса* (который, как мы видели, использует синтаксис инициализатора элемента), чтобы явно вызвать конструктор базового класса.

Конструкторы базового класса и операции присваивания базового класса не наследуются производными классами. Однако конструкторы и операции присваивания производного класса могут вызывать конструкторы и операции присваивания базового.

Конструктор производного класса всегда вызывает конструктор базового класса, чтобы сначала инициализировать элементы базового класса в производном классе. Если конструктор производного класса опущен, то конструктор по умолчанию производного класса вызывает конструктор базового класса. Деструкторы вызываются в порядке, обратном вызовам конструктора, поэтому деструктор производного класса вызывается прежде деструктора его базового класса.

Общее методическое замечание 19.3

Когда создается объект производного класса, сначала работает конструктор базового класса, затем конструкторы для объектов – элементов производного класса и, наконец, конструктор производного класса. Деструкторы вызываются в порядке, обратном вызовам соответствующих конструкторов.

Согласно стандарту C++ порядок, в котором создаются элементы-объекты, соответствует порядку, в котором они перечислены внутри определения класса. Порядок, в котором перечисляются инициализаторы элементов, не влияет на конструирование. При наследовании конструкторы базовых классов вызываются в том порядке, в котором указывается наследование в определении производного класса. Порядок, в котором указаны конструкторы базовых классов в конструкторе производного, не влияет на конструирование.

Распространенная ошибка программирования 19.4

Создание программы, зависящей от конкретного порядка конструирования элементов-объектов, может приводить к труднообнаружимым, тонким ошибкам.

Программа на рис. 19.6 показывает порядок, в котором вызываются конструкторы и деструкторы базового и производного классов. Программа состоит из пяти частей. Части 1 и 2 показывают простой класс **Point**, который состоит из конструктора, деструктора и защищенных элементов данных **x** и **y**. Конструктор и деструктор оба печатают объект **Point**, для которого они вызываются. Части 3 и 4 показывают простой класс **Circle**, полученный из класса **Point** по типу открытого наследования, в котором содержатся конструктор,

деструктор и элемент закрытых данных **radius** . Конструктор и деструктор оба печатают объект **Circle**, для которого вызваны. Конструктор **Circle**, кроме этого, активирует конструктор **Point** по синтаксису инициализатора элемента и передает значения для **a** и **b**, инициализируя таким образом элементы данных базового класса.

```
// POINT2.H
// Определение класса Point
#ifndef POINT2_H
#define POINT2_H

class Point {
public:
   Point(float = 0.0, float = 0.0);  // конструктор по умолчанию
   ~Point();                         // деструктор
protected:                  // доступны производным классам
   float x, y;              // x и y - координаты точки
};

#endif
```

Рис. 19.6. Определение класса **Point** (часть 1 из 5)

```
// POINT2.CPP
// Определения функций класса Point
#include <iostream.h>
#include "point2.h"

// Конструктор Point
Point::Point(float a, float b)
{
   x = a;
   y = b;

   cout << "Point constructor: "
        << '[' << x << ", " << y << ']' << endl;
}

// Деструктор Point
Point::~Point()
{
   cout << "Point destructor: "
        << '[' << x << ", " << y << ']' << endl;
}
```

Рис. 19.6. Определения функций-элементов класса **Point** (часть 2 из 5)

На рис. 19.6, часть 5 показан тестер для иерархии **Point** / **Circle**. Программа начинается с создания экземпляра объекта **Point** в его собственной области действия внутри **main**. Объект входит и немедленно выходит из области действия, поэтому сразу вызываются конструктор и деструктор. Затем программа создает экземпляр объекта **circle1** класса **Circle**. Это активирует конструктор **Point** с выводом на печать значений, переданных из конструктора **Circle**, затем выполняется печать, указанная в конструкторе **Circle**. После этого создается экземпляр **circle2** класса **Circle**. И снова вызываются оба конструктора **Point** и **Circle**. Обратите внимание, что тело конструктора **Point** выполняется

раньше, чем тело конструктора **Circle**. Достигается конец программы **main**, поэтому необходимо вызвать деструкторы для объектов **circle1** и **circle2**. Деструкторы вызываются в порядке, обратном вызовам соответствующих конструкторов. Поэтому деструкторы **Circle** и **Point** вызываются (именно в таком порядке) для объекта **circle2**, затем деструкторы **Circle** и **Point** вызываются в том же порядке для объекта **circle1**.

```
// CIRCLE2.H
// Определение класса Circle
#ifndef CIRCLE2_H
#define CIRCLE2_H

#include "point2.h"
#include <iomanip.h>

class Circle : public Point {
public:
   // конструктор по умолчанию
   Circle (float r = 0.0, float x = 0, float y = 0);

   ~Circle ();      // деструктор
private:
   float radius;   // радиус круга
};

#endif
```

Рис. 19.6. Определение класса **Circle** (часть 3 из 5)

```
// CIRCLE2.CPP
// Определения функций класса Circle
#include "circle2.h"

// Конструктор Circle вызывает конструктор Point
Circle::Circle(float r, float a, float b)
   : Point(a, b)   // вызов базового конструктора
{
   radius = r;

   cout << "Circle constructor: radius is "
        << radius << " [" << a << ", " << b << ']' << endl;
}

// Деструктор Circle
Circle::~Circle()
{
cout << "Circle destructor:  radius is "
     << radius << " [" << x << ", " << y << ']' << endl;
}
```

Рис. 19.6. Определения функций-элементов класса **Circle** (часть 4 из 5)

```
// FIG19_6.CPP
// Демонстрирует порядок вызова конструкторов
// и деструкторов базового и производного классов.
#include <iostream.h>
#include "point2.h"
#include "circle2.h"

main()
{
   // Показать вызовы конструкторов и деструкторов для Point
   {
      Point p(1.1, 2.2);
   }

   cout << endl;
   Circle circle1(4.5, 7.2, 2.9);
   cout << endl;
   Circle circle2(10, 5, 5);
   cout << endl;
   return 0;
}
```

```
Point constructor: [1.1, 2.2]
Point destructor: [1.1, 2.2]

Point  constructor: [7.2, 2.9]
Circle constructor: radius is 4.5 [7.2, 2.9]

Point  constructor: [5, 5]
Circle constructor: radius is 10 [5, 5]

Circle destructor: radius is 10 [5, 5]
Point  destructor: [5, 5]
Circle destructor: radius is 4.5 [7.2, 2.9]
Point  destructor: [7.2, 2.9]
```

Рис. 19.6. Порядок, в котором вызываются конструкторы и деструкторы базового и производного классов (часть 5 из 5)

19.10. Неявное преобразование объектов производного класса к базовому

Несмотря на то, что объект производного класса также «является» и объектом базового, типы объектов производного класса и базового класса различны. Объекты производного класса можно обрабатывать как объекты базового класса. Это имеет смысл, так как производный класс содержит элементы, соответствующие каждому из элементов базового класса, — как вы помните, обычно производный класс имеет больше элементов, чем базовый класс. В обратном направлении операция не имеет смысла, так как преобразование объекта базового класса в объект производного класса оставило бы дополнительные (по отношению к базовому классу) элементы неопределенными. Хотя такое преобразование «естественно» не допускается, ему можно придать смысл при наличии соответствующей перегруженной операции присваивания или конструктора преобразования.

Распространенная ошибка программирования 19.5

Присваивание объекта производного класса объекту соответствующего базового класса и затем попытка обращения к элементам, наличным только в производном классе.

Указатель на объект производного класса может быть неявно преобразован в указатель на объект базового класса, так как объект производного класса является объектом базового.

Существует четыре возможных способа для смешивания и согласования указателей базового и производного классов с объектами базового и производного классов:

1. Ссылка на объект базового класса с помощью указателя базового класса естественна.
2. Ссылка на объект производного класса с помощью указателя производного класса также естественна.
3. Ссылка на объект производного класса с помощью указателя базового класса безопасна, так как объект производного класса является также и объектом базового. Программа может ссылаться только на элементы базового класса. В случае, когда программа ссылается на элементы, наличные только в производном классе, с помощью указателя базового класса, компилятор сообщит о синтаксической ошибке.
4. Ссылка на объект базового класса с помощью указателя производного класса является синтаксической ошибкой. Необходимо сначала привести указатель производного класса к указателю базового класса.

Распространенная ошибка программирования 19.6

Приведение указателя базового класса к указателю производного класса может привести к ошибкам, если далее этот указатель используется для ссылки на объект базового класса, который не содержит требуемых элементов производного класса.

Как ни удобно иногда манипулировать объектами производного класса как объектами базового, причем делать это с помощью указателей базового класса, тут существует одна проблема. Например, в системе для платежных ведомостей было бы желательно обрабатывать связанный список сотрудников и рассчитывать еженедельную заработную плату для каждого персонально. Однако использование указателей базового класса позволяет программе вызывать только процедуру базового класса для расчета платежей (если такая процедура в базовом классе имеется). Нам нужен метод активации нужной процедуры расчета для каждого объекта, вне зависимости от того, является ли он объектом базового или производного класса, причем делать это просто с помощью указателя базового класса. Такое решение достигается посредством виртуальных функций и полиморфизма, как вы увидите в главе 20.

19.11. Наследование в конструировании программного обеспечения

Мы можем использовать механизм наследования для того, чтобы *настраивать*, т.е. максимально приспосабливать, существующее программное обес-

печение для конкретных целей. Мы наследуем свойства и поведение существующего класса, а затем вводим или удаляем какие-то атрибуты и поведение, чтобы получить нужный нам класс. В C++ можно сделать так, чтобы производный класс не имел доступа к исходному коду базового класса, однако для компоновки программы необходимо, чтобы он имел доступ к объектному коду. Эта замечательная возможность привлекательна для независимых разработчиков программного обеспечения. Они могут разрабатывать собственные классы для продажи или лицензирования и поставлять их пользователям в объектном формате. Пользователи смогут быстро получать новые классы из существующих библиотечных и в то же время не будут иметь доступа к исходному коду производителей. Все, что независимым разработчикам потребуется поставлять вместе с объектным кодом, — это заголовочные файлы.

Студентам трудно представить себе проблемы, с которыми сталкиваются проектировщики и разработчики больших программных проектов. Люди, искушенные в таких делах, постоянно утверждают, что ключом к совершенствованию процесса разработки является возможно более широкое повторное использование программного обеспечения. Объектно-ориентированное программирование вообще, и C++ в частности, способствуют такому подходу.

Имеются и доступны очень полезные и хорошие библиотеки классов, предоставляющие максимальные возможности для повторного использования программного кода. И поскольку интерес к C++ растет, то и интерес к библиотекам классов также будет расти. Точно так же, как с разработкой программного обеспечения, поставляемого независимыми проектировщиками и разработчиками, которая становится быстро развивающейся отраслью по мере распространения персональных компьютеров, будет обстоять дело и с созданием и продажей библиотек классов. Разработчики приложений будут строить их на основе библиотек классов, а проектировщики библиотек классов будут вознаграждаться за поставку своих библиотек с этими приложениями. Поставляемые в настоящее время с трансляторами библиотеки классов являются в основном библиотеками общего назначения и имеют ограниченную сферу применения. То, что мы увидим в ближайшее время — это всемирное массированное наступление в области разработки библиотек классов для огромного множества сфер приложений.

Общее методическое замечание 19.4

Создание производного класса не влияет на исходный или объектный код его базового класса; целостность базового класса при наследовании сохраняется.

Базовый класс описывает нечто обобщенное. Все классы, полученные из одного базового класса, наследуют его свойства. В процессе объектно-ориентированного проектирования разработчик ищет общность и выделяет ее в различных структурах, формируя базовые классы. Затем производные классы настраиваются — приобретают новые свойства, дополнительные к унаследованным от базового класса.

Точно так же, как проектировщик не ориентированных на объекты систем стремится избегать употребления ненужных функций, так и проектировщик объектно-ориентированных классов должен избегать разрастания ненужных классов. Такое разрастание классов создает проблемы управления ими и может мешать повторному использованию программного обеспечения просто потому, что усложняет поиск нужного класса в огромной библиотеке. Решение заключается в том, чтобы создавать меньше классов, снабжая каждый из них новыми существенными дополнительными функциональными свойствами.

Возможно, такие классы для определенных пользователей будут слишком насыщены потенциальными возможностями. В этом случае они могут маскировать эту избыточность, «принижая» таким образом свойства классов так, чтобы это соответствовало их задачам.

Совет по повышению эффективности 19.1

Если образованные путем наследования классы больше, чем требуется, то это может привести к ненужным расходам ресурсов памяти и снижению производительности.

Обратите внимание на то, что чтение набора определений производного класса может вводить в заблуждение из-за того, что унаследованные элементы не показаны, но тем не менее присутствуют в нем.

Общее методическое замечание 19.5

В объектно-ориентированной системе классы часто тесно связаны. «Вычеркните» общие атрибуты и модели поведения и поместите их в базовом классе, а затем используйте наследование для построения производных классов.

Общее методическое замечание 19.6

Производный класс содержит атрибуты и модели поведения своего базового класса. Он может также определять дополнительные атрибуты и поведение. Благодаря наследованию базовый класс можно компилировать независимо от производного класса. Необходимо только компилировать затем вновь введенные атрибуты и поведение, чтобы можно было их комбинировать с базовым классом для создания производного класса.

Общее методическое замечание 19.7

При модификации базового класса не требуется изменять производные классы, если только открытый интерфейс базового класса остается неизменным.

19.12. Композиция в сравнении с наследованием

Мы обсудили отношения типа *«является»*, они поддерживаются механизмом наследования. Мы также рассматривали отношения типа *«имеет»*, в них объект может содержать другие объекты в качестве элементов. Такие отношения создают новые классы путем *композиции* существующих классов. Например, заданы классы сотрудников, дней рождения и телефонных номеров — соответственно **Employee**, **BirthDate** и **TelephoneNumber**. Неправильно говорить, что **Employee** *«является»* днем рождения **BirthDate** или телефонным номером **TelephoneNumber**, а правильно, конечно, сказать, что сотрудник **Employee** *«имеет»* день рождения и телефонный номер — соответственно **BirthDate** и **TelephoneNumber**.

Вспомните, что порядок, в котором конструируются элементы, задается тем, в каком порядке эти объекты объявляются. Конструкторы базовых классов вызываются в порядке, в котором указывается наследование в производном классе.

Общее методическое замечание 19.8

При модификации класса элемента не требуется изменять его охватывающий класс, если только открытый интерфейс класса-элемента остается неизменным. Однако имейте в виду, что составной класс, возможно, потребуется перекомпилировать.

19.13. Отношения «использует» и «знает»

И наследование, и композиция способствуют повторному использованию программного обеспечения благодаря возможности создавать новые классы, имеющие много общего с существующими. Имеются и другие способы использовать возможности классов. Конечно, нельзя сказать, что некто *«является»* автомобилем, и объект, связанный с ним, не содержит автомобиль, но, конечно же, можно сказать, что субъект *«использует»* автомобиль. Функция использует объект, просто вызывая функции-элементы этого объекта

Объект может *«знать»* другой объект. Базы знаний часто имеют такие отношения. Один объект может содержать указатель или ссылку на другой объект, зная таким образом об этом объекте. В таком случае говорят что один объект находится с другим объектом в отношении «знания».

19.14. Пример: Point, Circle, Cylinder

Теперь перейдем к основному упражнению этой главы. Рассмотрим иерархию классов: **Point** — точка, **Circle** — окружность, **Cylinder** — цилиндр. Сначала создадим и применим класс **Point** (рис. 19.7), затем представим один пример, в котором выведем из класса **Point** класс **Circle** (рис. 19.8), и наконец, представим пример, в котором из класса **Circle** получается класс **Cylinder** (рис. 19.9).

Рис. 19.7, часть 1 показывает определение класса **Point**. Обратите внимание на то, что элементы данных защищены. Поэтому, когда мы получаем класс **Circle** из класса **Point**, функции-элементы класса **Circle** могут вызывать координаты **x** и **y** непосредственно, а не через функции доступа. Это может улучшить эффективность классов.

```
// POINT2.H
// Определение класса Point
#ifndef POINT2_H
#define POINT2_H

class Point {
   friend ostream &operator<<(ostream &, const Point &);
public:
   Point (float = 0, float = 0);       // конструктор по умолчанию
   void setPoint (float, float);       // установка координат
   float getX() const { return x; }     // получение координаты x
   float getY() const { return y; }     // получение координаты y

protected:          // доступны производным классам
   float x, y;        // координаты точки
};

#endif
```

Рис. 19.7. Определение класса **Point** (часть 1 из 3)

Рис. 19.7, часть 2 показывает определения функций-элементов класса **Point**. Заметьте, что программа **main** должна использовать функции доступа **getX** и **getY** для чтения значений защищенных элементов-данных **x** и **y**; помните,

что защищенные элементы данных доступны только для элементов и друзей их класса, а также для элементов и друзей производных от него классов.

```
// POINT2.CPP
// Функции-элементы Point
#include <iostream.h>
#include "point2.h"

// Конструктор Point
Point::Point(float a, float b)
{
   x = a;
   y = b;
}

// Установка координат x и y
void Point::setPoint(float a, float b)
{
   x = a;
   y = b;
}

// Вывод точки
   ostream &operator<<(ostream &output, const Point &p)
{
   output << '[' << p.x << ", " << p.y " ']';
   return output;          // разрешает конкатенацию
}
```

Рис. 19.7. Определения функций-элементов класса **Point** (часть 2 из 3)

```
// FIG19_7.CPP
// Тестер для класса Point
#include <iostream.h>
#include "point2.h"

main()
{
   Point p(7.2, 11.5);   // создать объект p класса Point

   // защищенные данные Point недоступны в main
   cout << "X coordinate is " << p.getX()
        << "\nY coordinate is " << p.getY();

   p.setPoint(10, 10);
   cout << "\n\nThe new location of p is " << p << endl;
   return 0;
}
```

```
X coordinate is 7.2
Y coordinate is 11.5

The new location of p is [10, 10]
```

Рис. 19.7. Тестер для класса **Point** (часть 3 из 3)

Следующий наш пример показан на рис. 19.8. Определения класса **Point** и его функций-элементов из рис. 19.7 здесь используются повторно. Рис. 19.7, части 1–3 показывают соответственно определение класса **Circle**, функций-элементов класса **Circle** и программу-тестер. Обратите внимание на то, что класс **Circle** выводится из **Point** по типу открытого наследования. Это означает, что открытый интерфейс класса **Circle** включает в себя функции-элементы класса **Point**, а также функции-элементы класса **Circle** — **setRadius**, **getRadius** и **area**. Заметьте, что перегруженная функция передачи в поток для **Circle** может непосредственно ссылаться на элементы **x** и **y**, потому что они являются защищенными элементами базового класса. Заметьте также, что необходимо ссылаться на элементы **x** и **y** через объект, используя нотацию **c.x** и **c.y** — перегруженная функция-операция помещения в поток не является элементом класса **Circle**. Она только дружественна ему. Программа-тестер создает экземпляр объекта класса **Circle**, затем использует ***get*-функции**, чтобы получить информацию об объекте класса **Circle**. Опять же, поскольку программа main не только не функция-элемент класса **Circle**, но и не дружественна к нему, она не может прямо ссылаться на защищенные данные класса **Circle**. Затем тестер использует ***set*** функции: **setRadius** и **setPoint**, чтобы переопределить радиус и координаты центра окружности. Наконец, тестер выполняет нечто особенно интересное. Он инициализует переменную-указатель **pRef** типа «ссылки на объект **Point**» **(Point &)** объектом **c** класса **Circle**. Программа затем печатает переменную **pRef**, которая, несмотря на тот факт, что она инициализирована объектом **Circle**, «думает», что это объект **Point**, в результате объект **Circle** фактически печатается как объект **Point**.

```
// CIRCLE2.H
// Определение класса Circle
#ifndef CIRCLE2_H
#define CIRCLE2_H

#include "point2.h"

class Circle : public Point {
   friend ostream &operator<<(ostream &, const Circle &);
public:
   // конструктор по умолчанию
   Circle(float r = 0.0, float x = 0, float y = 0);
   void setRadius(float);       // установка радиуса
   float getRadius() const;     // получение радиуса
   float area() const;          // вычисление площади
protected;          // доступны в производных классах
   float radius;    // радиус круга
};

#endif
```

Рис. 19.8. Определение класса **Circle** (часть 1 из 3)

Наш последний пример показан на рис. 19.9. Определения классов **Circle** и **Point**, определения их функций-элементов, приведенные на рис. 19.7 и 19.8, используются здесь повторно. Части 1–3 показывают соответственно определения класса **Cylinder**, его функций-элементов и программу-тестер. Обратите внимание на то, что класс **Cylinder** выводится из **Circle** по типу открытого наследования. Это означает, что открытый интерфейс класса **Cylinder** включает в себя функции-элементы класса **Circle**, а также функции-элементы класса

Cylinder — setHeight, getHeight, area (переопределяет функцию класса **Circle**) и **volume**. Заметьте, что перегруженная функция-операция передачи в поток объекта **Cylinder** может непосредственно ссылаться на элементы **x** и **y**, потому что они являются защищенными элементами базового класса **Circle** (**x** и **y** наследуются классом **Circle** от класса **Point**). Заметьте также, что необходимо ссылаться на элементы **x**, **y** и **radius**, используя нотацию **c.x**, **c.y**, **c.radius**, так как перегруженная функция-операция помещения в поток не является функцией-элементом класса **Cylinder**, а только дружественна ему. Тестер создает экземпляр объекта класса **Cylinder**, затем вызывает ***get***-функции, чтобы получить информацию об объекте **Cylinder**. Снова, поскольку **main** не только не функция-элемент класса **Cylinder**, но и не дружественна ему, она не может прямо ссылаться на защищенные данные класса **Cylinder**. Затем тестер использует ***set***-функции **setHeight, setRadius** и **setPoint**, чтобы переопределить высоту, радиус и координаты цилиндра. Наконец, тестер инициализует переменную-указатель **pRef** типа «ссылки на объект **Point**» (**Point &**) объектом **cyl** класса **Cylinder**. Затем программа печатает переменную **pRef**, которая, несмотря на тот факт, что она инициализирована объектом **Cylinder**, «думает», что это объект **Point**, в результате объект **Cylinder** фактически печатается как объект **Point**. После этого переменная-указатель **cRef** типа «ссылки на объект класс **Circle**» (**Circle &**) инициализуется объектом **cyl** класса **Cylinder**. Тестер затем печатает переменную **cRef**, которая, несмотря на тот факт, что она инициализирована объектом **Cylinder**, «думает», что это объект класса **Circle**, в результате объект **Cylinder** фактически печатается как объект класса **Circle**. На печать также выводится площадь объекта класса **Circle**.

```
// CIRCLE2.CPP
// Определения функций-элементов Circle
#include <iostream.h>
#include <iomanip.h>
#include "circle2.h"

// Конструктор Circle вызывает конструктор Point
// с инициализатором элемента и инициализирует радиус
Circle::Circle(float r, float a, float b)
   : Point (a, b)       // вызов базового конструктора
{ radius = r; }
// Set radius
void Circle::setRadius(float r) { radius = r; }
// Получение радиуса
float Circle: :getRadius() const { return radius; }

// Вычисление площади круга
float Circle::area() const
   { return 3.14159 * radius * radius; }

// Вывод круга в форме:
// Center = [x, y]; Radius = #.##
ostream &operator<<(ostream &output, const Circle &c)
{
   output << "Center = [" << c.x << ", " << c.y
          << "]; Radius = " << setiosflags(ios::showpoint)
          << setprecision(2) << c.radius;
   return output;    //  Разрешеает конкатенацию вызовов
}
```

Рис. 19.8. Определения функций-элементов класса **Circle** (часть 2 из 3)

```
// FIG19_8.CPP
// Тестер для класса Circle
#include <iostream.h>
#include "point2.h"
#include "circle2.h"

main()
{
   Circle c(2.5, 3.7, 4.3);

   cout << "X coordinate is " << c.getX()
        << "\nY coordinate is " << c.getY()
        << "\nRadius is " << c.getRadius();

   c.setRadius(4.2 5 ) ;
   c.setPoint(2, 2);
   cout << "\n\nThe new location and radius of c are\n"
        << c << "\nArea " << c.area() << endl;

   Point &pRef = c;
   cout << "\circle printed as a Point is: "
        << pRef << endl;

   return 0;
}
```

```
X coordinate is 3.7
Y coordinate is 4.3
Radius is 2.5
The new location and radius of c are
Center = [2, 2]; Radius = 4.25
Area: 56.74
Circle printed as a Point is: [2.00, 2.00]
```

Рис. 19.8. Тестер для класса **Circle** (часть 3 из 3)

```
// CYLINDR2.H
// Определение класса Cylinder
#ifndef CYLINDR2_H
#define CYLINDR2_H
#include "circle2.h"

class Cylinder : public Circle {
   friend ostream& operator<<(ostream&, const Cylinders);
public:
   // конструктор по умолчанию
   Cylinder (float h = 0.0, float r = 0.0,
             float x = 0.0, float y = 0.0);
   void setHeight (float);    // установка высоты
   float getHeight() const;   // получение высоты
   float area() const;        // вычисление и возврат площади
   float volume() const;      // вычисление и возврат объема
protected:
   float height;              // высота цилиндра
};

#endif
```

Рис. 19.9. Определение класса **Cylinder** (часть 1 из 3)

Пример прекрасно демонстрирует открытое наследование, а также определение и ссылку на защищенные элементы данных. Читатель теперь должен иметь хорошее представление об основах механизма наследования. В следующей главе мы покажем, как программировать в общем виде с помощью иерархий наследования, используя полиформизм. Абстракция, наследование и полиморфизм составляют суть объектно-ориентированного программирования.

```
// CYLINDR2.CPP
// Определение функций-элементов класса Cylinder
#include <iostream.h>
#include <iomanip.h>
#include "cylindr2.h"

// Конструктор Cylinder вызывает конструктор Circle
Cylinder::Cylinder(float h, float r, float x, float y)
   : Circle (r, x, y)   // вызов базового конструктора
{ height = h; }

// Установка высоты цилиндра
void Cylinder::setHeight(float h) { height = h; }

// Получение высоты цилиндра
float Cylinder::getHeight() const { return height; }

// Вычисление площади цилиндра (т.е. площади поверхности)
float Cylinder::area() const
{
   return 2 * Circle::area() +
          2 * 3.14159 * radius * height;
}

// Вычисление объема цилиндра
   float Cylinder::volume() const
   { return Circle::area() * height; }

// Вывод размеров цилиндра
ostream& operator<<(ostream &output, const Cylinders c)
{
   output << "Center = [" << c.x << ", " << c.y
          << "]; Radius = " << setiosflags(ios::showpoint)
          << setprecision(2) << c.radius
          << "; Height = " << c.height;
   return output;   // разрешает конкатенацию вызовов
}
```

Рис. 19.9. Определения функций-элементов и дружественной функции класса **Cylinder** (часть 2 из 3)

19.15. Сложное наследование

До сих пор в этой главе мы обсуждали механизм простого наследования, в котором каждый класс выводится только из одного базового класса. Однако класс может быть получен из нескольких базовых классов; такой механизм образования класса называется *сложным наследованием*. Сложное наследование означает, что производный класс наследует элементы нескольких базовых

классов. Этот мощный механизм дает интересные формы повторного использования программного обеспечения, но вместе с тем может порождать ряд проблем, связанных с неоднозначностью.

```
// FIG19_9.CPP
// Тестер для класса Cylinder
#include <iostream.h>
#include <iomanip.h>
#include "po±nt2.h"
#include "circle2.h"
#include "cylindr2.h "

main()
{
   // создать объект Cylinder
   Cylinder cyl(5.7, 2.5, 1.2, 2.3);

   // Вызов get-функций для вывода Cylinder
   cout << "X coordinate is " << cyl.getX()
        << "\nYcoordinate is " << cyl.getY()
        << "\nRadius is " << cyl.getRadius()
        << "\nHeight is " << cyl.getHeight();

   // Вызов set-функций для изменения атрибутов Cylinder
   cyl. setHeight (10) ;
   cyl. setRadius (4.25);
   cyl. setPoint (2, 2) ;
   cout << "\n\nThe new location, radius, "
        << "and height of cyl are:\n" << cyl << endl << "Area: "
        << cyl.area() << "   Volume: " << cyl.volume();

   // Вывести Cylinder как Point
   Point &pRef = cyl;   // pRef "думает" что это Point
   cout << "\n\nCylinder printed as a Point is: " << pRef;

   // Вывести Cylinder как Circle
   Circle &cRef = cyl;  // cRef "думает" что это Circle
   cout << "\n\nCylinder printed as a Circle is:\n" << cRef
        << "\nArea: " << cRef.area() << endl;
   return 0;
}
```

```
X coordinate is 1.2
Y coordinate is 2.3
Radius is 2.5
Height is 5.7

The new location, radius, and height of cyl are:
Center = [2, 2]; Radius = 4.25; Height = 10.00
Area: 380.53   Volume: 567.45

Cylinder printed as a Point is: [2.00, 2.001
Cylinder printed as a Circle is;
Center s [2.00, 2.00); Radius = 4.25
Area: 56.74
```

Рис. 19.9. Тестер для класса **Cylinder** (часть 3 из 3)

Хороший стиль программирования 19.2

Сложное наследование — мощный механизм при правильном его использовании. Его следует применять, когда между новым типом и двумя или более типами существует отношение «является» (т.е. тип А является как типом В, так и типом С).

Рассмотрим пример сложного наследования, показанный на рис. 19.10. Класс **Base1** содержит один защищенный элемент — **value** целого типа, а также один конструктор, который устанавливает **value**, и открытую функцию-элемент **getData**, которая считывает **value**.

```
//  BASE1.H
// Определение класса Base1
#ifndef BASE1_H
#define BASE1_H

class Base1 {
public:
   Base1 (int x) { value = x; }
   int getData() const { return value; }
protected:          // доступно производным классам
   int value;       // наследуется производным классом
};

#endif
```

Рис. 19.10. Определение класса **Base1** (часть 1 из 6)

Класс **Base2** аналогичен классу **Base1** во всем, за исключением того, что его защищенные данные представляют собой переменную **letter** символьного типа **char**. Класс **Base2** тоже имеет открытую функцию-элемент **getData**, но эта функция считывает значение символьной переменной **letter**.

```
// BASE2.H
// Определение класса Base2
#ifndef BASE2_H
#define BASE2_H

class Base2 {
public:
   Base2 (char c) { letter = c; }
   char getData() const { return letter; }
protected:          // доступно производным классам
   char letter;     // наследуется производным классом
};

#endif
```

Рис. 19.10. Определение класса **Base2** (часть 2 из 6)

Класс **Derived** выводится из обоих классов **Base1** и **Base2** путем сложного наследования. Он имеет закрытый элемент данных **real** типа **float** и открытую функцию-элемент **getReal**, которая читает значение **real**.

Обратите внимание, как обозначается сложное наследование с помощью двоеточия (:) после class **Derived** — просто списком открытых базовых классов, разделенных запятыми. Заметьте также, что конструктор производного класса **Derived** вызывает каждый из своих базовых конструкторов, **Base1** и **Base2**, посредством синтаксиса инициализатора элементов.

```
// DERIVED.H
// Определение класса Derived, который наследует
// два базовых класса (Base1 и Base2).
#ifndef DERIVED_H
#define DERIVED_H

#include <iostream.h>
#include "base1.h"
#include "base2.h"

// сложное наследование
class Derived : public Base1, public Base2 {
   friend ostream &operator<<(ostream &, const Derived &);
public:
   Derived(int, char, float);
   float getReal() const;
private:
   float real;    // закрытые данные производного класса
};

#endif
```

Рис. 19.10. Определение класса **Derived** (часть 3 из 6)

```
// DERIVED.CPP
// Объявления функций-элементов класса Derived
#include "derived.h"

// Конструктор Derived вызывает конструкторы для
// классса Base1 и класса Base2.
Derived::Derived(int i, char c, float f)
      : Base1(i), Base2(c)  // вызов обоих базовых конструкторов
{ real = f; }

// Возврат значения real
float Derived::getReal() const { return real; }

// Вывести все данные Derived
ostream &operator<<(ostream &output, const Derived &d)
{
     output << "    Integer: " << d.value
            << "\n  Character: " << d.letter
            << "\nReal number: " << d.real;

return output;    // разрешает конкатенацию вызовов
}
```

Рис. 19.10. Определения функций-элементов класса **Derived** (часть 4 из 6)

Перегруженная операция передачи в поток для класса **Derived** использует нотацию с точкой для производного объекта **d**, чтобы напечатать переменные **value**, **letter** и **real**. Функция этой операции является дружественной классу **Derived**, поэтому **operator<<** может иметь прямой доступ к закрытому элементу данных **real** класса **Derived**. Кроме того, так как эта операция является дружественной производному классу, она может иметь доступ к защищенным элементам **value** и **letter** соответствующих базовых классов **Base1** и **Base2**.

```
// FIG19_10.CPP
// Тестер для примера сложного наследования
#include <iostream.h>
#include "base1.h"
#include "base2.h"
#include "derived.h"

main()
{
   Base1 b1 (10), *base1Ptr;    // создать объект базового класса
   Base2 b2('z'), *base2Ptr;    // создать объект другого базового
                                // класса
   Derived d(7, 'A', 3.5);      // создать объект производного
                                // класса

   // напечатать данные базовых классов
   cout << "Object b1 contains integer    "
        << b1.getData()
        << "\nObject b2 contains character "
        << b2.getData()
        << "\nObject d contains:\n" << d;

   // напечатать данные производного класса
   // операция разрешения области действия устраняет
   // неоднозначность getData
   cout << "\n\nData members of Derived can be"
        << " accessed individually:\n"
        << "    Integer: " << d.Base1::getData()
        << "\n  Character: " << d.Base2::getData()
        << "\nReal number: " << d.getReal() << "\n\n";

   cout << "Derived can be treated as an "
           << "object of either base class:\n";

   // рассматривает Derived как объект Base1
   base1Ptr = &d;
   cout << "base1Ptr->getData() yields "
        << base1Ptr->getData();

   // рассматривает Derived как объект Base2
   base2Ptr = &d;
   cout << "\nbase2Ptr->getData() yields "
        << base2Ptr->getData() << endl;
   return 0;
}
```

Рис. 19.10. Пример для тестирования сложного наследования (часть 5 из 6)

Содержимое каждого из объектов базовых классов печатается при статическом их связывании. Несмотря даже на то, что существуют две функции **getData**, вызовы будут однозначными, поскольку они указывают сразу на версию **getData** объекта **b1** и версию **getData** объекта **b2**.

Затем мы печатаем содержимое объекта **d** класса **Derived**, связывание здесь также статическое. Но в этом случае действительно возникает проблема неоднозначности, поскольку этот объект содержит две функции **getData** — одну, унаследованную от класса **Base1**, другую — от класса **Base2**. Эта неоднозначность легко снимается с помощью операции разрешения области действия

(т.е. применяется нотация **d.Base1::getData()**, чтобы напечатать целое значение в **value** и **d.Base2::getData()**, чтобы напечатать символьное в **letter**. Значение с плавающей запятой в **real** печатается однозначно, поскольку есть вызов **d.getReal()**.

Далее покажем, что отношения простого наследования *является* применимы также и к сложному наследованию. Мы присваиваем адрес производного объекта **d** указателю **base1Ptr** базового класса и печатаем целое значение в **value**, вызывая функцию-элемент **getData()** через указатель **base1Ptr**. Затем присваиваем адрес производного объекта **d** указателю базового класса **base2Ptr** и печатаем символьное значение в **letter**, вызывая функцию-элемент **getData** через указатель **base2Ptr**.

```
Object b1 contains integer 10
Object b2 contains character z
Object d contains:
            Integer: 7
          Character: A
  Real number: 3.5
Data members of Derived can be accessed individually:
            Integer: 7
          Character: A
 Real number: 3.5
Derived can be treated as an object of either base class:
base1Ptr->getData() yields 7
base2Ptr->getData() yields A
```

Рис. 19.10. Пример для тестирования сложного наследования (часть 6 из 6)

Этот пример показал механику сложного наследования и дал представление об одной из простых проблем неоднозначности. Сложное наследование представляет собой непростой механизм, подробнее оно рассмотрено в учебниках по C++, а также в нашей книге *«C++: How to Program»* (Prentice-Hall, 1994)[1].

Резюме

- Одним из ключевых аспектов объектно-ориентированного программирования является повторное использование программного обеспечения посредством наследования.
- Программист может указать, что новый класс должен наследовать элементы данных и функции-элементы ранее определенного базового класса. В этом случае новый класс называется производным классом.
- При простом наследовании новый класс выводится только из одного базового класса. При сложном наследовании новый производный класс наследует нескольким (возможно, никак не связанным между собой) базовым классам.
- Как правило, производный класс вводит свои собственные элементы данных и функции-элементы, так что, вообще говоря, производный класс имеет более обширное определение, чем его базовый класс. Про-

1 Русский перевод: Х.М. Дейтел, П.Дж. Дейтел «Как программировать на C++», БИНОМ, М., 1999. — *Прим. ред.*

изводный класс содержит больше специфических свойств, чем базовый, и обычно представляет меньшую группу объектов.

- Производный класс не может иметь доступа к закрытым элементам своего базового класса: это нарушило бы инкапсуляцию последнего. Однако производный класс имеет доступ к открытым и защищенным элементам своего базового класса.
- Конструктор производного класса всегда вызывает конструктор своего базового класса, чтобы сначала создать и инициализировать элементы базового класса в объекте производного класса.
- Деструкторы вызываются в порядке, обратном вызовам конструкторов, поэтому деструкторы производного класса вызываются раньше деструкторов своего базового класса.
- Наследование можно рассматривать как инструмент повторного использования программного обеспечения, позволяющий сберегать время на разработку программ и способствующий применению уже отлаженного и апробированного программного кода высокого качества.
- Наследование позволяет применять существующие библиотеки классов.
- Когда-нибудь программное обеспечение будет в основном строиться из стандартных повторно используемых компонентов точно так же, как конструируется в настоящее время аппаратная часть компьютеров.
- Разработчику программного обеспечения не обязательно иметь доступ к исходному коду базового класса, нужно только знать интерфейс базового класса и иметь для компоновки его объектный код.
- Объект производного класса можно обрабатывать как объект соответствующего открытого базового класса, однако обратное неверно.
- Базовый класс находится в иерархическом отношении с производным классом.
- Один класс может существовать сам по себе. Когда этот класс участвует в наследовании, он становится либо базовым классом, который поставляет атрибуты и поведение другим классам, либо производным классом с унаследованными свойствами и поведением базовых классов.
- Иерархия наследования может иметь любую глубину в рамках физических пределов какой-то частной системы.
- Иерархии являются полезными инструментами для понимания и управления сложными структурами. Что касается все возрастающей сложности программного обеспечения, то C++ предусматривает поддержку иерархических структур посредством механизмов наследования и полиморфизма.
- Для преобразования указателя базового класса к указателю производного можно использовать явное приведение. Если такой указатель должен быть разыменован, то сначала следует убедиться, что он указывает на объект производного класса.
- Защищенный доступ выполняет роль промежуточного уровня защиты между открытым и закрытыми доступами. К защищенным элементам базового класса имеют доступ функции-элементы и друзья самого класса, а также элементы и друзья классов, производных от данного; никакие другие функции не могут обращаться к защищенным элементам класса.

- Защищенные элементы используются для расширения привилегий производных классов, причем эти привилегии не распространяются на не-дружественные функции и функции других классов.
- Сложное наследование приводит к графоподобным иерархиям. Эти графы не имеют циклов, потому что все стрелки указывают в одном и том же направлении. Такой граф называется направленным ациклическим графом (DAG).
- При выводе нового класса из базового класса последний можно объявить как открытый, защищенный либо закрытый.
- При выводе нового класса из открытого базового класса открытые элементы базового класса становятся открытыми элементами производного класса, а защищенные элементы базового класса становятся защищенными элементами производного.
- При выводе нового класса из защищенного базового класса открытые и защищенные элементы базового класса становятся защищенными элементами производного класса.
- При выводе нового класса из закрытого базового класса открытые и защищенные элементы базового класса становятся закрытыми элементами производного класса.
- Базовый класс может быть либо непосредственным базовым классом производного класса, либо косвенным базовым классом производного класса. Непосредственный базовый класс явно указывается при объявлении производного класса. Косвенный базовый класс явно не указывается; он располагается на несколько уровней выше по древовидной иерархической структуре.
- Если элемент базового класса не подходит для производного класса, его можно просто переопределить в производном классе.
- Важно различать отношения «является» и «имеет». В отношении «имеет» объект одного класса содержит объект другого класса. В отношении «является» объект типа производного класса можно также рассматривать как объект типа базового класса. Отношение «является» — это наследование. Отношение «имеет» — композиция.
- Объект производного класса можно присваивать объекту базового класса. Такое присваивание имеет смысл, так как производный класс имеет элементы, однозначно соответствующие элементам базового класса.
- Указатель на объект производного класса может быть неявно преобразован в указатель на объект базового класса.
- Можно преобразовать указатель базового класса в указатель производного класса с помощью явного приведения. Указатель должен ссылаться на объект производного класса.
- Базовый класс определяет общность. Все классы, производные от него, наследуют свойства этого базового класса. В объектно-ориентированном процессе проектирования разработчик ищет общее и выделяет это общее, чтобы создать осмысленные базовые классы. Производные классы затем могут быть настроены и снабжены свойствами, выходящими за рамки унаследованных от базового класса.

- Чтение ряда объявлений производного класса может вводить в заблуждение, так как не все элементы производного класса присутствуют в этих объявлениях. В частности, унаследованные элементы не перечисляются в объявлениях производного класса, хотя на самом деле они в нем имеются.
- Отношение «имеет» является примером создания новых классов на основе композиции существующих классов.
- Конструкторы элементов-объектов вызываются в порядке, в котором элементы объявлены. При наследовании конструкторы базовых классов вызываются в порядке, в котором указано наследование, и перед конструктором производного класса.
- Для объекта производного класса сначала вызывается конструктор базового класса, затем — конструктор производного класса.
- Когда объект производного класса выходит из области действия, деструкторы вызываются в порядке, обратном конструкторам — сначала вызывается деструктор производного класса, затем деструктор базового.
- Один класс можно вывести из нескольких базовых классов. Это называется сложным наследованием.
- Сложное наследование обозначается двоеточием (:) с последующим списком разделенных запятыми базовых классов.
- Конструктор производного класса вызывает конструкторы для каждого из своих базовых классов посредством синтаксиса инициализатора элементов.

Терминология

абстракция
базовый класс
библиотеки классов
деструктор базового класса
деструктор производного класса
друзья базового класса
друзья производного класса
закрытое наследование
закрытый базовый класс
защищенное наследование
защищенный базовый класс
защищенный элемент класса
иерархическое отношение
иерархия классов
инициализатор базового класса
класс элемента
ключевое слово **protected**
композиция
конструктор базового класса
конструктор по умолчанию базового класса
конструктор производного класса
косвенный базовый класс
надкласс
направленный ациклический граф
наследование
настройка программного обеспечения
неоднозначность в сложном наследовании
непосредственный базовый класс
Объектно-ориентированное программирование
объект-элемент
открытое наследование
открытый базовый класс
отношение *знает*
отношение *имеет*
отношение *использует*
отношение *является*
ошибка бесконечной рекурсии
переопределение элемента базового класса
повторное использование программного кода
подкласс
пользователь класса
производный класс

простое наследование
сложное наследование
стандартизованные компоненты программного обеспечения
указатель базового класса
указатель на объект базового класса
указатель на объект производного класса
указатель производного класса
управление доступом к элементам

Распространенные ошибки программирования

19.1. Использование объекта базового класса как объекта производного класса.

19.2. Использование элементов производного класса, не определенных в базовом классе, после преобразования указателя объекта базового класса к указателю на объект производного класса.

19.3. Когда функция-элемент базового класса переопределяется в производном классе, функция производного класса обычно вызывает одноименную функцию базового и выполняет некоторую дополнительную работу. В случае, когда для вызова функции базового класса не используется операция разрешения области действия, возникает бесконечная рекурсия, так как функция-элемент производного класса фактически вызывает саму себя.

19.4. Создание программы, зависящей от конкретного порядка конструирования элементов-объектов, может приводить к труднообнаружимым, тонким ошибкам.

19.5. Присваивание объекта производного класса объекту соответствующего базового класса и затем попытка обращения к элементам, наличным только в производном классе.

19.6. Приведение указателя базового класса к указателю производного класса может привести к ошибкам, если далее этот указатель используется для ссылки на объект базового класса, который не содержит требуемых элементов производного класса.

Хороший стиль программирования

19.1. Когда в производном классе наследуются ненужные свойства, маскируйте их, переопределяя функции.

19.2. Сложное наследование — мощный механизм при правильном его использовании. Его следует применять, когда между новым типом и двумя или более типами существует отношение «является» (т.е. тип A является как типом B, так и типом C)

Совет по повышению эффективности

19.1. Если образованные путем наследования классы больше, чем требуется, то это может привести к ненужным расходам ресурсов памяти и снижению производительности.

Общие методические замечания

19.1. Производный класс не может иметь прямого доступа к закрытым элементам своего базового класса.

19.2. Переопределение функции-элемента в производном классе не требует одинаковой сигнатуры с функцией базового класса.

19.3. Когда создается объект производного класса, сначала работает конструктор базового класса, затем конструкторы для объектов — элементов производного класса и, наконец, конструктор производного класса. Деструкторы вызываются в порядке, обратном вызовам соответствующих конструкторов.

19.4. Создание производного класса не влияет на исходный или объектный код его базового класса; целостность базового класса при наследовании сохраняется.

19.5. В объектно-ориентированной системе классы часто тесно связаны. «Вычеркните» общие свойства и модели поведения и поместите их в базовом классе, а затем используйте наследование для построения производных классов.

19.6. Производный класс содержит атрибуты и модели поведения своего базового класса. Он может также определять дополнительные атрибуты и поведение. Благодаря наследованию базовый класс можно компилировать независимо от производного класса. Необходимо только компилировать затем вновь введенные атрибуты и поведение, чтобы можно было их комбинировать с базовым классом для создания производного класса.

19.7. При модификации базового класса не требуется изменять производные классы, если только открытый интерфейс базового класса остается неизменным.

19.8. При модификации класса элемента не требуется изменять его охватывающий класс, если только открытый интерфейс класса-элемента остается неизменным. Однако имейте в виду, что составной класс, возможно, потребуется перекомпилировать.

Упражнения для самоконтроля

19.1. Заполните пропуски в следующих утверждениях:

a) Если класс **Alpha** наследует классу **Beta**, то класс **Alpha** называется _____ классом, а класс **Beta** _____ классом.

b) C++ реализует механизм _____, который позволяет производному классу наследовать от нескольких базовых классов, даже если эти базовые классы не связаны между собой.

c) Наследование позволяет _____, что укорачивает время разработки, а также обеспечивает возможность использования уже проверенного высококачественного программного кода.

d) Объект _____ класса можно рассматривать как объект соответствующего _____ класса.

e) Чтобы преобразовать указатель базового класса в указатель производного класса, следует использовать _____, так как компилятор считает это опасной операцией.

f) Три спецификатора доступа к элементам — это _____, _____ и _____.

g) При выведении нового класса с открытым наследованием открытые элементы базового класса становятся _____ элементами производного класса, а защищенные элементы базового класса становятся _____ элементами производного класса.

h) При выведении нового класса из одного базового класса с защищенным наследованием открытые элементы базового класса становятся _____ элементами производного класса, а защищенные элементы базового класса стаовятся _____ элементами производного класса.

i) Отношение «имеет» между классами представлено _____, а отношение «является» представляется механизмом _____.

Ответы на упражнения для самоконтроля

19.1. a) производным, базовым; b) сложного наследования; c) повторно использовать программное обеспечение; d) производного, базового; e) приведение типа; f) **public**, **protected**, **private**; g) открытыми, защищенными; h) защищенными, защищенными; i) композицией, наследования.

Упражнения

19.2. Рассмотрим класс велосипедов — **Bicycle**. Используя ваше знание некоторых общих деталей велосипедов, покажите иерархию классов, в которой класс **Bicycle** наследует другим классам, которые, в свою очередь, наследуют еще некоторым классам. Обсудите создание различных представителей класса **Bicycle**. Обсудите наследование класса **Bicycle** для других, тесно связанных с ним, производных классов.

19.3. Кратко определите каждый из следующих терминов: наследование, сложное наследование, базовый класс и производный класс.

19.4. Обсудите, почему преобразование базового указателя в указатель производного класса компилятор считает опасным.

19.5. Выпишите все формы, которые вы можете придумать — как двумерные, так и трехмерные, — и постройте из них иерархию форм. Иерархия должна иметь базовый класс **Shape**, которому наследуют классы **TwoDimensionalShape** и **ThreeDimensionalShape**. После того, как вы получили иерархию, определите каждый из ее классов. Мы используем эту иерархию в главе 20, чтобы обрабатывать все формы как объекты базового класса **Shape**. Эта методика называется полиморфизмом.

19.6. Часто используются классы, называемые коллекционными или контейнерными. Такие классы хранят элементы других классов. Типичные разновидности коллекционных классов — массивы, стеки, связанные списки, деревья, символьные строки, множества, портфели, словари, таблицы и т.д. Коллекционный класс, как правило, предусматривает такие действия, как «вставить элемент»,

«исключить элемент», «найти элемент», «соединить две коллекции», «определить пересечение двух коллекций» (т.е. найти их общие элементы), «напечатать коллекцию», «найти наибольший элемент», «найти наименьший элемент», «найти сумму элементов» и т.д.

a) Напишите список всех типов классов коллекций, которые вы сможете придумать (в том числе упомянутые здесь списки).

b) Организуйте из этих классов коллекций иерархию. Вы, возможно, захотите разделить их на упорядоченные и неупорядоченные коллекции.

c) Реализуйте столько классов, сколько сможете, используя наследование, минимизирующее число нового кода, который придется написать для создания новых классов.

d) Напишите тестер, который проверяет каждый из классов в вашей иерархии наследования.

глава



Виртуальные функции и полиморфизм



Цели

- Познакомиться с понятием полиморфизма.
- Понять, как реализуется полиморфизм при объявлении и использовании виртуальных функций.
- Понять различие между конкретными и абстрактными классами.
- Изучить объявление чисто виртуальных функций для создания абстрактных классов.
- Оценить важность полиморфизма для разработки и сопровождения расширяемых систем.

Содержание

20.1. Введение
20.2. Обработка различных типов данных при помощи операторов switch
20.3. Виртуальные функции
20.4. Абстрактные базовые классы и конкретные классы
20.5. Полиморфизм
20.6. Пример: программа начисления заработной платы, использующая возможности полиморфизма
20.7. Новые классы и динамическое связывание
20.8. Виртуальные деструкторы
20.9. Пример: наследование интерфейса и реализации

Резюме • *Распространенные ошибки программирования* • *Хороший стиль программирования* • *Советы по повышению эффективности* • *Общие методические замечания* • *Упражнения для самоконтроля* • *Ответы на упражнения для самоконтроля* • *Упражнения*

20.1. Введение

Благодаря механизму *виртуальных функций* и *полиморфизму* стало возможным проектировать и реализовывать системы, которые легко расширять. Такие программы могут обрабатывать объекты существующих классов и классов, которых не существует на момент разработки программы. Если эти классы получены на основе известных программе классов, то, поскольку программа обеспечивает механизм обработки объектов базовых классов, она сможет успешно обрабатывать и объекты производных классов. Многие мощные программные продукты построены на этом принципе.

20.2. Обработка различных типов данных при помощи операторов switch

Один из способов обработки объектов различных типов — использование оператора **switch**. Однако при этом возникает ряд проблем. Программист может забыть сделать проверку типа в нужном месте программы. Он может выполнить в операторе **switch** обработку не всех типов данных. При добавлении нового типа данных программист может забыть вставить новые **case**-метки в операторы **switch**. И наконец, поскольку при добавлении нового типа данных изменения должны быть внесены во все операторы **switch**, это займет много времени и появятся новые ошибки.

Как мы скоро увидим, механизм виртуальных функций и полиморфизма позволяет отказаться от логики программирования, основанной на операторе **switch**. Механизм виртуальных функций вносит необходимую логику автоматически. При этом не возникает ошибок, свойственных прежней методике.

Общее методическое замечание 20.1

Интересным следствием использования виртуальных функций и полиморфизма является простота и ясность программ. В них становится меньше фрагментов с ветвлением и больше обычного линейного кода. Такие программы проще тестировать, отлаживать и сопровождать.

20.3. Виртуальные функции

Предположим, что следующий набор классов геометрических фигур: **Circle**, **Triangle**, **Rectangle**, **Square** и т.д. выведен из базового класса **Shape**. В объектно-ориентированном программировании каждый из этих классов может быть наделен способностью нарисовать самого себя. Каждый класс имеет собственную функцию **draw** и эти функции существенно различаются. При рисовании любого такого объекта было бы удобно трактовать его как объект базового класса **Shape**. Тогда, чтобы нарисовать любую фигуру, мы могли бы просто вызывать функцию **draw** базового класса **Shape**, переложив на программу динамическое (т.е. времени выполнения) определение того, функцию какого производного класса требуется вызвать.

Чтобы сделать возможным такой способ поведения, мы объявляем функцию **draw** в базовом классе как *виртуальную функцию* и затем *переопределяем* ее в каждом из производных классов функцией, которая может нарисовать соответствующую фигуру. Функция *объявляется виртуальной при помощи* ключевого слова **virtual** в прототипе функции в базовом классе.

Общее методическое замечание 20.2

Если функция была объявлена как виртуальная, она остается виртуальной во всей иерархии наследования.

Общее методическое замечание 20.3

Если в производном классе виртуальная функция не определяется, то он просто наследует функцию своего непосредственного класса-предшественника.

Хороший стиль программирования 20.1

Хотя функция производного класса является виртуальной, если на высших уровнях иерархии было сделано соответствующее объявление, многие программисты, для придания ясности программе, предпочитают использовать явное определение функции как виртуальной на всех уровнях иерархии.

Если мы вызываем функцию **draw**, которая объявлена в базовом классе как виртуальная, и используем для этой цели указатель на базовый класс, ссылающийся на объект производного класса, то программа выберет динамически (т.е. во время выполнения) функцию draw нужного производного класса. Это называется *динамическим связыванием* (см. рис. 20.1 и рис. 20.2).

Общее методическое замечание 20.4

Переопределяемая виртуальная функция должна иметь тот же возвращаемый тип и список параметров, что и виртуальная функция базового класса.

Распространенная ошибка программирования 20.1

Определение в производном классе виртуальной функции, не совпадающей по типу возвращаемого значения или списку параметров с функцией в базовом классе, приводит к синтаксической ошибке.

Если виртуальная функция вызывается при помощи ссылки на объект по его имени и операции-точки выбора элемента класса, то ссылка разрешается во время компиляции (это называется *статическим связыванием*) и вызывается виртуальная функция из того класса (определенная в нем или унаследованная), которому принадлежит данный объект.

Перегрузка функций не использует динамическое связывание. Перегрузка разрешается во время компиляции подбором функции с сигнатурой, соответствующей вызову (возможно, с неявным приведением типов). Это пример статического связывания.

20.4. Абстрактные базовые классы и конкретные классы

Когда мы говорим о классе как о типе, мы подразумеваем, что будут создаваться объекты этого типа. Однако во многих случаях бывает полезно определять классы, объекты которых не будут создаваться. Такие классы называются *абстрактными классами*. Поскольку они используются в наследовании в качестве базовых классов, мы будем называть их *абстрактными базовыми классами*. Объекты абстрактного базового класса создавать нельзя.

Единственная цель определения абстрактного класса состоит в том, чтобы предусмотреть обобщенный базовый класс, на основе которого строится иерархия наследования. Классы, для которых могут создаваться объекты, называются *конкретными классами*.

Например, мы могли бы определить абстрактный базовый класс **TwoDimensionalObject** и произвести от него классы типа **Square**, **Circle**, **Triangle** и т.д. Мы могли бы также создать абстрактный базовый класс **ThreeDimensio-**

nalObject и получить от него конкретные классы типа **Cube**, **Sphere**, **Cylinder**, **Pyramid** и т.д. Эти абстрактные базовые классы слишком обобщенны для того, чтобы из них можно было создавать реальные объекты; нужно внести в них какую-то конкретику, прежде чем мы сможем говорить о создании объектов. Конкретные классы имеют специфические особенности, благодаря которым имеет смысл создавать объекты таких классов.

Класс, имеющий виртуальные функции, становится абстрактным, если одна или несколько его виртуальных функций объявлены чистыми. *Чистая виртуальная функция* — это функция, объявление которой завершается инициализатором **= 0**, как в следующем примере:

```
virtual float earnings() const = 0;  // чистая виртуальная функция
```

Общее методическое замечание 20.5

Если в классе, базовый класс которого содержит чистую виртуальную функцию, эта функция не определяется, то она остается чистой и в этом (производном) классе. Как следствие, производный класс сам является абстрактным классом.

Распространенная ошибка программирования 20.2

Попытка создать объект абстрактного класса (т.е. содержащего одну или несколько чистых виртуальных функций) приводит к синтаксической ошибке.

Иерархия классов не должна в обязательном порядке содержать абстрактные классы, но, как мы увидим, многие хорошо продуманные объектно-ориентированные системы имеют иерархии классов, возглавляемые абстрактным базовым классом. В некоторых случаях абстрактные классы занимают несколько верхних уровней иерархии. Хороший пример такой иерархии — иерархия геометрических фигур. Такая иерархия могла бы возглавляться абстрактным базовым классом **Shape**. На следующем уровне мы можем произвести еще два абстрактных базовых класса, а именно **TwoDimensionalShape** и **ThreeDimensionalShape**. И на третьем уровне иерархии можно определять конкретные классы двумерных фигур вроде кругов и квадратов и конкретные классы трехмерных форм вроде сфер и кубов.

20.5. Полиморфизм

Понятие *полиморфизма* в C++ означает способность объектов различных классов, связанных наследованием, различным образом реагировать на вызов одной и той же функции-элемента. Если, например, класс **Rectangle** выведен из класса **Quadrilateral**, то объект **Rectangle** — более специфическая форма объекта **Quadrilateral**. Действие (например, вычисление периметра или площади), которое может быть выполнено над объектом класса **Quadrilateral**, может также выполняться на объекте класса **Rectangle**.

Полиморфное поведение обеспечивается механизмом виртуальных функций. При вызове виртуальной функции через указатель (или ссылку) на базовый класс C ++ выбирает переопределенную функцию правильного производного класса, соответствующего объекту, на который ссылается указатель.

В производном классе может переопределяться и не-виртуальная функция-элемент базового класса. Если такая функция вызывается через указа-

тель на базовый класс, компилятор генерирует вызов функции базового класса. Если функция-элемент вызывается при помощи указателя на производный класс, генерируется вызов функции производного класса. Это пример неполиморфного поведения.

Использование виртуальных функций и полиморфизма позволяет одному и тому же вызову функции-элемента выполнять различные действия, зависящие от типа объекта, получившего вызов. Этот механизм предоставляет программисту «выразительные средства», которые трудно переоценить. В следующих разделах главы мы увидим примеры мощи полиморфизма и виртуальных функций.

Общее методическое замечание 20.6

Благодаря полиморфизму и виртуальным функциям программист может сосредоточить свое внимание на общих задачах, переложив частные вопросы на среду исполнения. Программист может управлять большим набором объектов, каждый из которых имеет свое специфическое поведение, не зная даже типов этих объектов.

Общее методическое замечание 20.7

Полиморфизм обеспечивает расширяемость системы: программный код, опирающийся на полиморфное поведение объектов, не зависит от типа объектов, которым передается вызов. Таким образом, без всяких изменений основной системы в нее может быть введен новый тип объектов, который может реагировать на существующий набор сообщений. Программы не придется перекомпилировать, за исключением, возможно, некоторой части клиента, в которой создаются объекты нового типа.

Общее методическое замечание 20.8

В абстрактном классе определяется интерфейс всех узлов иерархии классов. Абстрактный класс содержит чистые виртуальные функции, которые будут определятся в производных классах. Благодаря полиморфизму этот интерфейс класса будет доступен всем функциям в иерархии.

Хотя мы не можем создавать объекты абстрактных базовых классов, мы можем объявлять указатели на абстрактные базовые классы. Такие указатели могут затем использоваться в полиморфных манипуляциях с объектами производных конкретных классов, когда такие объекты будут созданы.

Давайте рассмотрим пример полиморфизма и виртуальных функций в действии. Это будет некий экранный менеджер, который выводит на экран разнообразные объекты, включая объекты тех типов, что могут быть введены в систему уже после создания программы — менеджера. Системе может потребоваться выводить на экран различные геометрические формы (т.е. базовым классом здесь является класс **Shape**), как, например, квадраты, окружности, треугольники, прямоугольники и другие фигуры (каждый из этих классов фигур выводится из базового класса **Shape**). Менеджер экрана для работы с объектами всех этих классов использует указатели на базовый класс. Для того, чтобы нарисовать любой объект (независимо от уровня иерархии классов, в котором этот объект появляется), менеджер использует указатель на базовый класс, ссылающийся на нужный объект, и передает этому объекту сообщение **draw**. Функция **draw** объявлена чистой виртуальной в базовом классе **Shape** и определяется в каждом из производных классов. Каждый объект знает, как

нарисовать самого себя. Экранный менеджер не должен беспокоиться относительно этих деталей, он просто передает каждому объекту сообщение «нарисовать самого себя».

Полиморфизм особенно эффективен при создании многоуровневых систем программного обеспечения. В операционных системах, например, каждый тип физического устройства может функционировать совершенно отлично от устройств других типов. Независимые от устройства команды *read* или *write*, читающие или записывающие данные с устройств и на устройства, являются командами универсальными. Команда *write*, посланная объекту «драйвер устройства», должна интерпретироваться этим драйвером в соответствии с тем, каким образом он управляет устройствами конкретного типа. Но сам по себе вызов команды *write* не зависит от устройства — это просто команда передать указанному устройству некоторое число байт данных из памяти. Объектно-ориентированная операционная система может использовать абстрактный базовый класс, чтобы обеспечить соответствующий интерфейс для всех драйверов устройств. Все производные классы наследуют общее поведение этого абстрактного базового класса. Возможные действия драйверов устройств (т.е. общий интерфейс) определяются чистыми виртуальными функциями абстрактного базового класса. Реализация этих виртуальных функций, соответствующих специфическим типам драйверов устройств, осуществляется в производных классах.

В главе 17 мы ввели понятие итераторов. Общепринятым является определение *класса-итератора*, который может осуществлять обход всех объектов в совокупности. Если вы хотите напечатать список объектов в связанном списке, то вы, например, можете создать объект класса-итератора, который будет при своем вызове возвращать следующий элемент связанного списка. Итераторы обычно используются в полиморфном программировании при обходе связанного списка из объектов различных уровней иерархии. В таком списке все указатели были бы указателями на базовый класс.

20.6. Пример: программа начисления заработной платы, использующая возможности полиморфизма

Давайте применим виртуальные функции и полиморфизм в программе начисления зарплаты, которое производится с учетом типа служащего (рис. 20.1). Мы будем использовать базовый класс **Employee**. Мы создадим следующие производные от **Employee** классы: **Boss** — работники, получающие фиксированное еженедельное жалованье, независимо от количества отработанных часов; **CommissionWorker** — сотрудники, которые получают фиксированный оклад плюс процент от продаж; **PieceWorker** — рабочие, жалованье которых определяется количеством единиц произведенной продукции, и **HourlyWorker** — те, кто имеют почасовую оплату.

Ко всем служащим, разумеется, должен быть применим вызов функции начисления зарплаты **earnings**. Но способ, которым вычисляется доход каждого сотрудника, определяется классом, к которому он принадлежит, и все эти классы являются производными от базового класса **Employee**. Поэтому функция **earnings** объявляется виртуальной в базовом классе **Employee**, а соответствующие ее реализации вводятся в каждом производном классе. Затем, при расчете дохода любого сотрудника, программа вызывает функцию **earnings**

при помощи указателя на базовый класс, ссылающегося на соответствующий объект производного класса. В настоящей программе расчета заработной платы объекты, соответствующие типам работников, могли бы храниться в связанном списке с указателями типа **Employee ***. И тогда программа просто проходила бы по узлам этого списка, используя указатели **Employee ***, и вызывала бы функцию earnings каждого объекта.

Давайте теперь рассмотрим класс **Employee** (рис. 20.1, части 1 и 2). В открытой секции класса объявляются следующие функции-элементы: конструктор именем и фамилией работника в качестве аргументов; деструктор, который освобождает динамически выделенную память; get-функция, которая возвращает имя сотрудника; get-функция, возвращающая фамилию; и наконец, две чистых виртуальных функции — **earnings** и **print**. Почему эти функции — чистые виртуальные? Ответ прост: потому что не имеет смысла определять эти функции в классе **Employee**. Объявляя эти функции чистыми виртуальными, мы указываем на то, что будем определять эти функции в производных классах, но не в базовом классе. Мы не можем рассчитать доход для служащего вообще — сначала нужно знать категорию этого служащего.

```
// EMPLOY2.H
// Абстрактный базовый класс Employee
#ifndef EMPLOY2_H
#define EMPLOY2_H

class Employee {
public:
   Employee(const char *, const char *);
   ~Employee();
   const char *getFirstName() const;
   const char *getLastName() const;

   // Чистые виртуальные функции превращают класс
   // Employee в абстрактный базовый класс.
   virtual float earnings() const = 0; // чистая виртуальная
                                       // функция
   virtual void print() const = 0;     // чистая виртуальная
                                       // функция
private:
   char *firstName;
   char *lastName;
};

#endif
```

Рис. 20.1. Абстрактный базовый класс **Employee** (часть 1 из 12)

Класс **Boss** (рис. 20.1, части 3 и 4) выведен из класса **Employee** с открытым наследованием. Открытые функции-элементы этого класса включают в себя конструктор, который имеет параметры: имя, фамилию и еженедельное жалованье, и передает имя и фамилию конструктору класса **Employee**, который присваивает значения элементам-данным **firstName** и **lastName** (наследуемым производными классами); set-функцию, присваивающую новое значение закрытому элементу данных **weeklySalary**; виртуальную функцию **earnings** вычисления дохода служащего **Boss**; виртуальную функцию **print**, которая выводит категорию служащего и его имя.

```
// EMPLOY2.CPP
// Определение функций-элементов
// абстрактного базового класса Employee.
//
// Замечание: чистые виртуальные функции не определяются
#include <iostream.h>
#include <string.h>
#include <assert.h>
#include "employ2.h"

// Конструктор динамически выделяет память для
// имени и фамилии и при помощи функции strcpy
// копирует имя и фамилию в соответствующие элементы данных.
Employee::Employee(const char *first, const char *last)
{
   firstName = new char[ strlen(first) + 1 ];
   assert(firstName != 0);    // проверка выделения памяти
   strcpy(firstName, first);

   lastName = new char[ strlen(last) + 1 ];
   assert(lastName != 0);   // проверка выделения памяти
   strcpy(lastName, last);
}

// Деструктор освобождает динамически выделенную память
Employee::~Employee()
{
   delete [] firstName;
   delete [] lastName;
}

// Возвращает указатель на имя сотрудника
const char *Employee::getFirstName() const
{
   // const не позволяет вызываемой функции изменять закрытые
   // данные. Вызывающая функция должна скопировать возвращаемую
   // строку, прежде чем деструктор удалит динамически выделенную
   // память и значение указателя станет неопределенным.

   return firstName;    // после вызова память должна быть
                        // освобождена
}

// Возвращает указатель на фамилию сотрудника
const char *Employee::getLastName() const
{
   // const не позволяет вызываемой функции изменять закрытые
   // данные. Вызывающая функция должна скопировать возвращаемую
   // строку, прежде чем деструктор удалит динамически выделенную
   // память и значение указателя станет неопределенным.

   return lastName;    // после вызова память должна быть
                       // освобождена
}
```

Рис. 20.1. Определения функций-элементов абстрактного базового класса **Employee** (часть 2 из 12)

```
// BOSS1.H
// класс Boss, производный от класса Employee
#ifndef BOSS1_H
#define BOSS1_H
#include "employ2.h"

class Boss : public Employee {
public:
   Boss(const char *, const char *, float = 0.0);
   void setWeeklySalary(float);
   virtual float earnings() const;
   virtual void print() const;
private:
   float weeklySalary;
};

#endif
```

Рис. 20.1. Класс **Boss**, производный от абстрактного базового класса **Employee** (часть 3 из 12)

```
// BOSS1.CPP
// Определение функций-элементов класса Boss
#include <iostream.h>
#include  "boss1.h"

// Конструктор класса Boss
Boss::Boss(const char *first, const char *last, float s)
   : Employee(first, last)  // вызов конструктора базового класса
{ weeklySalary = s > 0 ? s : 0; }

// Задание жалованья работника типа Boss
void Boss::setWeeklySalary(float s)
{ weeklySalary = s > 0 ? s : 0; }

// Возвращает сумму, начисленную работнику типа Boss
float Boss::earnings() const { return weeklySalary; }

// Вывод имени и фамилии Boss'a
void Boss::print() const
{
   cout << "\n               Boss: " << getFirstName()
        << ' ' <<   getLastName();
}
```

Рис. 20.1. Определение функций-элементов класса **Boss** (часть 4 из 12)

Класс **CommissionWorker** (рис. 20.1, части 5 и 6) является производным базового класса **Employee** с открытым наследованием. Открытые функции-элементы класса включают: конструктор, который имеет в качестве параметров имя, фамилию, жалованье, комиссию и количество проданных товаров и передает имя и фамилию конструктору класса **Employee** для инициализации наследуемых данных-элементов базового класса **firstName** и **lastName**; set-функции, присваивающие новые значения закрытым элементам данных **salary**, **commission** и **quantity**; виртуальную функцию **earnings**, определяющую порядок начисления жалованья работника **CommissionWorker**, и виртуальную функцию **print**, которая выводит категорию служащего, его имя и фамилию.

```
// COMMIS1.H
// Класс CommissionWorker, производный от класса Employee
#ifndef COMMIS1_H
#define COMMIS1_H
#include "employ2.h"

class CommissionWorker : public Employee {
public:
   CommissionWorker(const char *, const char *,
                    float = 0.0, float = 0.0, int = 0);
   void setSalary(float);
   void setCommission(float);
   void setQuantity(int);
   virtual float earnings() const;
   virtual void print() const;
private:
   float salary;      // недельный оклад
   float commission; // комиссионные за единицу проданного товара
   int quantity;      // количество проданного за неделю товара
};

#endif
```

Рис. 20.1. Класс **CommissionWorker**, производный от абстрактного базового класса **Employee** (часть 5 из 12)

```
// COMMIS1.CPP
// Определения функций-элементов класса CommissionWorker
#include <iostream.h>
#include "commis1.h"

// Конструктор класса CommissionWorker
CommissionWorker::CommissionWorker(const char *first,
   const char *last, float s, float c, int q)
   : Employee(first, last)  // вызов конструктора базового класса
{
   salary = s > 0 ? s : 0;
   commission = c > 0 ? c : 0;
   quantity = q > 0 ? q : 0;
}

// Устанавливает работнику CommissionWorker недельный оклад
void CommissionWorker::setSalary(float s)
{ salary = s > 0 ? s : 0; }

// Устанавливает работнику CommissionWorker комиссионные
void CommissionWorker::setCommission(float c)
{ commission = c > 0 ? c : 0; }

// Задает количество проданного работником
// CommissionWorker товара
CommissionWorker::setQuantity(int q)
{ quantity = q > 0 ? q : 0; }

// Вычисляет заработок работника CommissionWorker
CommissionWorker::earnings() const
{ return salary + commission * quantity; }
```

```
// Выводит имя и фамилию работника CommissionWorker
void CommissionWorker::print() const
{
   cout << "\nCommission worker: " << getFirstName()
        << ' ' << getLastName();
}
```

Рис. 20.1. Определение функций-элементов класса **CommissionWorker** (часть 6 из 12)

Класс **PieceWorker** (рис. 20.1, части 7 и 8) выведен из класса **Employee** с открытым наследованием. Открытые функции-элементы этого класса — конструктор с параметрами: имя, фамилия, плата за единицу произведенной продукции и количество произведенной продукции, который передает имя и фамилию конструктору класса **Employee** для инициализации наследуемых из базового класса элементов-данных **firstName** и **lastName**; set-функции, используемые для присвоения значений закрытым элементам данных **wagePerPiece** и **quantity**; виртуальная функция **earnings** начисления заработка работнику типа **PieceWorker** и виртуальная функция **print**, которая выводит тип служащего, его имя и фамилию.

```
// PIECE1.H
// Класс PieceWorker, производный от класса Employee
#ifndef PIECE1_H
#define PIECE1_H
#include "employ2.h"

class PieceWorker : public Employee {
public:
   PieceWorker(const char *, const char *,
               float = 0.0, int = 0);
   void setWage(float);
   void setQuantity(int);
   virtual float earnings() const;
   virtual void print() const;

private:
   float wagePerPiece;    // оплата за единицу
                          // произведенной продукции
   int quantity;          // произведено за неделю
};

#endif
```

Рис. 20.1. Производный класс **PieceWorker** от абстрактного базового класса **Employee** (часть 7 из 12)

Класс **HourlyWorker** (рис. 20.1, части 9 и 10) является производным от класса **Employee** с открытым наследованием. Открытыми функциями-элементами этого класса являются — конструктор с параметрами: имя, фамилия, ставка почасовой оплаты и число отработанных часов, передающий имя и фамилию конструктору класса **Employee**, который присваивает значения наследуемым данным-элементам **firstName** и **lastName**; set-функции, присваивающие значения закрытым элементам данных **wage** и **hours**; виртуальная функция начисления заработка **earnings** для рабочего типа **HourlyWorker**; виртуальная функция **print**, которая выводит тип служащего и его имя и фамилию.

```
// PIECE1.CPP
// Определение функций-элементов класса PieceWorker

#include <iostream.h>
#include "piece1.h"

// Конструктор класса PieceWorker
PieceWorker::PieceWorker(const char *first, const char *last,
                         float w, int q)
   : Employee(first, last)  // вызов конструктора базового класса
{
   wagePerPiece = w > 0 ? w : 0;
   quantity = q > 0 ? q : 0;
}

// Установка ставки оплаты за единицу продукции
void PieceWorker::setWage(float w)
{ wagePerPiece = w > 0 ? w : 0; }

// Задание количества произведенного товара
void PieceWorker::setQuantity(int q)
{ quantity = q > 0 ? q : 0; }

// Начисление заработка рабочему PieceWorker
float PieceWorker::earnings() const
{ return quantity * wagePerPiece; }

// Вывод имени и фамилии рабочего PieceWorker
void PieceWorker::print() const
{
   cout << "\n     Piece worker: " << getFirstName()
        << ' ' << getLastName();
}
```

Рис. 20.1. Определение функций-элементов класса **PieceWorker** (часть 8 из 12)

```
// HOURLY1.H
// Определение класса HourlyWorker
#ifndef HOURLY1_H
#define HOURLY1_H
#include "employ2.h"

class HourlyWorker : public Employee {
public:
   HourlyWorker(const char *, const char *,
                float = 0.0, float = 0.0);
   void setWage(float);
   void setHours(float);
   virtual float earnings() const;
   virtual void print() const;
private:
   float wage;              // оплата за час
   float hours;             // часы, отработанные за неделю
};

#endif
```

Рис. 20.1. Класс **HourlyWorker**, производный от абстрактного базового класса **Employee** (часть 9 из 12)

```
// HOURLY1.CPP
// Определение функций-элементов класса HourlyWorker
#include <iostream.h>
#include "hourly1.h"

// Конструктор класса HourlyWorker
HourlyWorker::HourlyWorker(const char *first, const char *last,
                           float w, float h)
   : Employee(first, last) // вызов конструктора базового класса
{
   wage = w > 0 ? w : 0;
   hours = h >= 0 && h < 168 ? h : 0;
}

// Установка ставки почасовой оплаты
void HourlyWorker::setWage(float w)
{ wage = w > 0 ? w : 0; }

// Задание количества отработанных часов
void HourlyWorker::setHours(float h)
{ hours = h >= 0 && h < 168 ? h : 0; }

// Вычисление заработка работника типа HourlyWorker
float HourlyWorker::earnings() const
{ return wage * hours; }

// Вывод имени и фамилии работника типа HourlyWorker
void HourlyWorker::print() const
{
   cout << "\n    Hourly worker: " << getFirstName()
        << ' ' << getLastName();
}
```

Рис. 20.1. Определение функций-элементов класса **HourlyWorker** (часть 10 из 12)

Программа-тестер (рис. 20.1, части 11 и 12) начинается с объявления указателя **ptr** на базовый класс типа **Emplopee ***. Все три последующих фрагмента кода функции **main** подобны друг другу, так что мы обсудим только первый фрагмент, в котором обрабатывается объект **Boss**. В строке

```
Boss b ("John", "Smith", 800.00);
```

создается объект **b** производного класса **Boss** с необходимыми конструктору аргументами, включая имя, фамилию и фиксированное жалованье. В строке

```
ptr = &b; // указателю на базовый класс присваивается адрес
          // объекта производного класса
```

указателю **ptr** на базовый класс присваивается адрес объекта **b** производного класса. Это действие мы должны выполнить обязательно, если хотим добиться полиморфного поведения. Строка

```
ptr->print();     // динамическое связывание
```

вызывает функцию-элемент **print** объекта, на который ссылается **ptr**. Поскольку в базовом классе функция **print** объявлялась виртуальной, будет вызвана функция **print** производного класса — такое поведение мы и называем полиморфным. Этот вызов функции — пример динамического связывания, когда функция вызывается через указатель на базовый класс, и решение, какую функцию вызывать, откладывается до времени исполнения программы. В строке

```
cout << " earned $" << ptr->earnings(); // динамическое связывание
```

производится вызов функции-элемента **earnings** объекта, на который ссылается **ptr**. Поскольку в базовом классе **earnings** была объявлена виртуальной функцией, во время выполнения программы будет вызываться функция **earnings** производного класса, соответствующего обрабатываемому объекту. Это был еще один пример динамического связывания. В строке

```
b.print();                                  // статическое связывание
```

явно вызывается функция-элемент **print** класса **Boss** применением операции выбора элемента (точки) к объекту **b** класса **Boss**. Это пример статического связывания, поскольку тип объекта, для которого вызывается функция, известен во время компиляции. Этот вызов функции включен как доказательство того, что при динамическом связывании вызывается правильная функция **print**. В строке

```
cout << " earned $" << b.earnings();     // статическое связывание
```

явно вызывается функция-элемент **earnings** класса **Boss** применением операции-точки к объекту **b** класса **Boss**. Это также пример статического связывания. Этот вызов функции используется для доказательства того, что при динамическом связывании вызывается правильная функция **earnings**.

```
// FIG20_1.CPP
// Программа формирования иерархии классов
#include <iostream.h>
#include <iomanip.h>
#include "employ2.h"
#include "boss1.h"
#include "commis1.h"
#include "piece1.h"
#include "hourly1.h"

main()
{
   // Установка параметров форматирования вывода
   cout  <<  setiosflags(ios::showpoint) << setprecision(2);

   Employee *ptr;  // указатель на базовый класс

   Boss b("John", "Smith", 800.00);
   ptr = &b;  // указателю на базовый класс присваивается адрес
              // объекта производного класса
   ptr->print();                        // динамическое связывание
   cout << " earned $" << ptr->earnings();
                                        // динамическое связывание
   b.print();                           // статическое связывание
   cout << " earned $" << b.earnings();
                                        // статическое связывание

   CommissionWorker c("Sue", "Jones", 200.0, 3.0, 150);
   ptr = &c;  // указателю на базовый класс присваивается адрес
              // объекта производного класса
   ptr->print();                        // динамическое связывание
```

Рис. 20.1. Иерархия классов, производных от абстрактного базового класса **Employee** (часть 11 из 12)

```
   cout << " earned $" << ptr->earnings();
                                        // динамическое связывание
   c.print();                           // статическое связывание
   cout << " earned $" << c.earnings();
                                        // статическое связывание

   PieceWorker p("Bob", "Lewis", 2.5, 200);
   ptr = &p;  // указателю на базовый класс присваивается адрес
              // объекта производного класса
   ptr->print();                        // динамическое связывание
   cout << " earned $" << ptr->earnings();
                                        // динамическое связывание
   p.print();                           // статическое связывание
   cout << " earned $" << p.earnings();
                                        // статическое связывание

   HourlyWorker h("Karen", "Price", 13.75, 40);
   ptr = &h;  // указателю на базовый класс присваивается адрес
              // объекта производного класса
   ptr->print();                        // динамическое связывание
   cout << " earned $" << ptr->earnings();
                                        // динамическое связывание
   h.print();                           // статическое связывание
   cout << " earned $" << h.earnings();
                                        // статическое связывание
   cout << endl;

   return 0;
}
```

```
            Boss:  John Smith earned $800.00
            Boss:  John Smith earned $800.00
Commission worker:  Sue Jones earned $650.00
Commission worker:  Sue Jones earned $650.00
     Piece worker:  Bob Lewis earned $500.00
     Piece worker:  Bob Lewis earned $500.00
    Hourly worker:  Karen Price earned $550.00
    Hourly worker:  Karen Price earned $550.00
```

Рис. 20.1. Иерархия классов, производных от абстрактного базового класса **Employee** (часть 12 из 12)

20.7. Новые классы и динамическое связывание

Полиморфизм и виртуальные функции безусловно будут работать, когда все возможные классы известны заранее. Но они также успешно работают и тогда, когда постоянно вводятся новые виды классов.

Новые классы адаптируются к системе при помощи динамического связывания (называемого также *поздним связыванием*). При вызове виртуальной функции тип объекта во время компиляции знать не нужно. Во время выполнения в такой вызов «подставляется» вызов функции-элемента класса, соответствующего обрабатываемому объекту.

Программа экранного менеджера может работать с новыми объектами, которые вводятся в систему, и ее не нужно каждый раз перекомпилировать. Все

вызовы функции **draw** остаются без изменений; новые объекты содержат функции рисования непосредственно в себе. В такую систему легко могут быть введены новые возможности, причем с минимальными затратами. И, как результат, расширяется область применения такой программы.

Динамическое связывание дает возможность независимым производителям программного обеспечения (ISV — Independent Software Vendors) распространять свои программы, не раскрывая являющихся их собственностью секретов. Распространяемое ими программное обеспечение может состоять только из файлов заголовков и объектных файлов. Исходные тексты можно не распространять. Разработчики, получившие такой продукт, могут затем, используя механизм наследования, вывести новые классы из полученных от ISV. Программное обеспечение, которое работает с поставляемыми ISV классами, будет работать и с классами, производными от них, вызывая (благодаря динамическому связыванию) замещающие виртуальные функции производных классов.

При динамическом связывании во время выполнения программы вызов виртуальной функции-элемента перенаправляется функции из соответствующего класса. Для этой цели используется *таблица виртуальных функций*, называемая *vtable*, которая реализована в виде массива, содержащего указатели на функции. Каждый класс, содержащий виртуальные функции, имеет таблицу *vtable*. Для каждой виртуальной функции класса в *vtable* включается указатель на функцию, используемую для объекта этого класса. Виртуальная функция, используемая в конкретном классе, может быть либо определена в нем, либо унаследована, непосредственно или косвенно, от стоящего выше в иерархии базового класса.

Определенная в базовом классе виртуальная функция-элемент может быть замещена в производном классе, но производные классы не должны делать это в обязательном порядке. Это значит, что производный класс может использовать виртуальную функцию-элемент из базового класса, что будет отражено в *vtable*.

Каждый объект класса с виртуальными функциями содержит указатель на *vtable* своего класса. Этот указатель для программиста недоступен. Соответствующий указатель на функцию выбирается из *vtable* и разыменовывается; на этом завершается формирование вызова виртуальной функции во время выполнения программы. На просмотр *vtable* и разыменование указателя требуются незначительные ресурсы системы.

Совет по повышению эффективности 20.1

Обеспечиваемый виртуальными функциями и динамическим связыванием полиморфизм довольно эффективен. Использование этих механизмов незначительно влияет на эффективность системы.

Совет по повышению эффективности 20.2

Полиморфизм, обеспечиваемый виртуальными функциями и динамическим связыванием, можно рассматривать как конкурента оператора **switch**. И здесь стоит заметить, что обычно оптимизирующие компиляторы C++ для программ, использующих полиморфизм, генерируют столь же эффективный код, как и код на основе логики оператора **switch**.

20.8. Виртуальные деструкторы

При использовании полиморфизма в работе с динамически выделяемыми объектами иерархии классов могут возникнуть трудности. Если объекты разрушаются при помощи операции **delete** с указателем на базовый класс, то вызывается деструктор базового класса. Это происходит независимо от типа объекта, на который ссылается указатель на базовый класс, и от того факта, что деструктор каждого класса имеет другое имя.

Имеется простое решение этой проблемы — деструктор базового класса объявляется виртуальным. Это приведет к тому, что деструкторы всех производных классов будут виртуальными, даже если их имена отличаются от имени деструктора базового класса. И теперь, если объект в иерархии разрушается при помощи операции **delete**, а указатель, ссылающийся на объект, является указателем на базовый класс, вызывается деструктор соответствующего производного класса.

Хороший стиль программирования 20.2

Если класс имеет виртуальные функции, определяйте деструктор также виртуальным, даже если это и не требуется в данном конкретном случае. Производные от него классы могут иметь свои деструкторы, и тогда вызываться будут именно они.

Распространенная ошибка программирования 20.3

Объявление конструктора виртуальной функцией. Конструктор не может быть виртуальным.

20.9. Пример: наследование интерфейса и реализации

В нашем следующем примере (рис. 20.2) мы вернемся к иерархии классов: точка, круг, цилиндр — из предыдущей главы, только на этот раз на вершину иерархии поставим абстрактный базовый класс **Shape**. **Shape** содержит чистую виртуальную функцию — **printShapeName** — и, следовательно, является абстрактным базовым классом. В **Shape** имеются две других виртуальных функции, а именно **area** и **volume**, каждая из которых возвращает нулевое значение. Класс **Point** наследует эти функции от **Shape**. И в этом есть смысл, потому что и площадь и объем точки равны нулю. Класс **Circle** наследует функцию **volume** от класса **Point**, но вводит свою реализацию функции **area**. Класс **Cylinder** имеет свои версии как функции **volume**, так и функции **area**.

Обратите внимание на то, что хотя **Shape** и является абстрактным базовым классом, в нем определяются некоторые функции-элементы, и реализации этих функций наследуются. Будем говорить, что три виртуальные функции класса **Shape** определяют наследуемый интерфейс, который будет присущ всем элементам иерархии. Кроме этого, класс **Shape** предусматривает некоторые реализации, которые наследуются производными классами в первых уровнях иерархии.

Этот пример подчеркивает, что класс может наследовать от базового класса интерфейс и реализацию. Иерархии, наследующие реализацию, ассоциируют свои функциональные свойства с верхними уровнями иерархии. Функциональные свойства иерархий, наследующих интерфейс, ассоциированы с более низкими уровнями.

Базовый класс **Shape** (рис. 20.2, часть 1) состоит из трех открытых виртуальных функций и не содержит данных. Функция **printShapeName** объявляется как чисто виртуальная, поэтому переопределяется в каждом из производных классов. Функции **area** и **volume** определены и возвращают значение **0.0**. Эти функции замещаются в тех производных классах, для которых изменяются формулы вычисления площади или объема.

```
// SHAPE.H
// Определение абстрактного базового класса Shape
#ifndef SHAPE_H
#define SHAPE_H

class Shape {
public:
   virtual float area() const { return 0.0; }
   virtual float volume() const { return 0.0; }
   virtual void printShapeName() const = 0; // чистая виртуальная
                                            // функция
};

#endif
```

Рис. 20.2. Определение абстрактного базового класса **Shape** (часть 1 из 9)

Класс **Point** (рис. 20.2, части 2 и 3) выведен из класса **Shape** открытым наследованием. Поскольку точка не имеет ни площади, ни объема, функции-элементы базового класса **area** и **volume** не замещаются, а наследуются от класса **Shape**. Здесь определяется функция **printShapeName**, которая была объявлена как чисто виртуальная в базовом классе. Другие функции-элементы включают в себя set-функцию, присваивающую новые значения **x** и **y** координатам объекта **Point**, и get-функции, возвращающие значения координат **x** и **y**.

```
// POINT1.H
// Определение класса Point
#ifndef POINT1_H
#define POINT1_H
#include <iostream.h>
#include "shape.h"

class Point : public Shape {
   friend ostream &operator << (oatream &, const Point &);
public:
   Point(float = 0, float = 0);  // конструктор по умолчанию
   void setPoint(float, float);
   float getX() const { return x; }
   float getY() const { return y; }
   virtual void printShapeName() const { cout << "Point:: "; }
private:
   float x, y;   // x и y координаты Point
};

#endif
```

Рис. 20.2. Определение класса **Point** (часть 2 из 9)

```
// POINT1.CPP
// Определение функций-элементов класса Point
#include <iostream.h>
#include "point1.h"

Point::Point(float a, float b)
{
   x=a;
   y=b;
}

void Point::setPoint(float a, float b)
{
   x=a;
   y=b;
}

ostream &operator << (ostream &output, const Point &p)
{
   output << '[' << p.x << ", " << p.y << ']';

   return output;   // разрешает последовательные вызовы
}
```

Рис. 20.2. Определения функций-элементов класса **Point** (часть 3 из 9)

Класс Circle (рис. 20.2, части 4 и 5) выводится из класса Point с открытым наследованием. Круг не имеет объема, поэтому функция-элемент volume базового класса не замещается, а наследуется из Shape (через класс Point). Функция area замещается, так как Circle имеет площадь. Функция printShapeName является реализацией чистой виртуальной функции, определенной в базовом классе. Если эту функцию не переопределить здесь, класс унаследует ее из класса Point.

```
// CIRCLE1.H
// Определение класса Circle
#ifndef CIRCLE1_H
#define CIRCLE1_H
#include "point1.h"

class Circle : public Point {
   friend ostream &operator << (ostream &, const Circle &);
public:
   // конструктор по умолчанию
   Circle(float r = 0.0, float x = 0.0, float y = 0.0);

   void setRadius(float);
   float getRadius() const;
   virtual float area() const;
   virtual void printShapeName() const { cout << "Circle: "; }
private:
   float radius;        // радиус Circle
};

#endif
```

Рис. 20.2. Определение класса **Circle** (часть 4 из 9)

```
// CIRCLE1.CPP
// Определение функций-элементов класса Circle
#include <iostream.h>
#include <iomanip.h>
#include "circle1.h"

// Конструктор класса Circle вызывает конструктор класса Point
Circle::Circle(float r, float a, float b)
   : Point(a, b)  // вызов конструктора базового класса
{  radius = r > 0 ? r : 0; }

// Задание значения радиуса
void Circle::setRadius(float r) { radius = r > 0 ? r : 0; }

// Получение значения радиуса
float Circle::getRadius() const { return radius; }

// Расчет площади круга
float Circle::area() const { return 3.14159 * radius * radius; }

// Вывод параметров круга в виде: Center=[x, y]; Radius=#.##
ostream &operator << (ostream &output, const Circle &c)
{
   output << '[' << c.getX() << ", " << c.getY()
          << "]; Radius = " << setiosflags(ios::showpoint)
          << setprecision(2) << c.radius;

   return output;   // разрешает последовательные вызовы
}
```

Рис. 20.2. Определение функций-элементов класса **Circle** (часть 5 из 9)

Set-функция присваивает новое значение радиуса объекту класса **Circle** и get-функция возвращает радиус **Circle**.

Класс **Cylinder** (рис. 20.2, части 6 и 7) выводится из класса **Circle** открытым наследованием. Поскольку **Cylinder** имеет площадь и объем, функции **area** и **volume** замещаются в этом классе. Функция **printShapeName** — реализация виртуальной функции, которая была определена как чистая виртуальная в базовом классе. Если эту функцию не заместить в этом классе, то он унаследует ее из класса **Circle**. Другая функция-элемент этого класса, set-функция, присваивает новое значение элементу данных **height** класса **Cylinder**.

Программа-тестер (рис. 20.2, части 8 и 9) начинается с создания объекта **point** класса **Point**, объекта **circle** класса **Circle** и объекта **cylinder** класса **Cylinder**. Далее вызывается функция **printShapeName** каждого объекта, которая выводит параметры объекта, используя перегруженную операцию помещения в поток вывода, как доказательство того, что объекты правильно инициализированы. Затем объявляется указатель **ptr** типа **Shape ***. Этот указатель используется для ссылки на производные от базового класса объекты. Сначала указателю **ptr** присваивается адрес **point** и производятся следующие вызовы:

```
ptr->printShapeName ()
ptr->area()
ptr->volume()
```

```
// CYLINDER.H
// Определение класса Cylinder
#ifndef CYLINDER_H
#define CYLINDER_H
#include "circle1.h"

class Cylinder : public Circle {
   friend ostream &operator << (ostream &, const Cylinder &);
public:
   // конструктор по умолчанию
   Cylinder(float h = 0.0, float r = 0.0,
            float x = 0.0, float y = 0.0);
   void setHeight(float);
   virtual float area() const;
   virtual float volume() const;
   virtual void printShapeName() const { cout << "Cylinder: "; }
private:
   float height;    // высота объекта Cylinder
};

#endif
```

Рис. 20.2. Определение класса **Cylinder** (часть 6 из 9)

```
// CYLINDR1.CPP
// Определения функций-элементов класса Cylinder
#include <iostream.h>
#include <iomanip.h>
#include  "cylindr1.h"

Cylinder::Cylinder(float h, float r, float x, float y)
: Circle(r, x, y)  // вызов конструктора базового класса
{ height = h > 0 ? h : 0; }

void Cylinder::setHeight(float h)
{ height = h > 0 ? h : 0; }

float Cylinder::area() const
{
   // площадь поверхности Cylinder
   return 2 * Circle::area() +
          2 * 3.14159 * Circle::getRadius() * height;
}

float Cylinder::volume() const
{
   float r = Circle::getRadius();
   return 3.14159 * r * r * height;
}

ostream &operator << (ostream &output, const Cylinder &c)
{
   output << '[' << c.getX() << ", " << c.getY()
          << "]; Radius = " << setiosflags(ios::showpoint)
          << setprecision(2) << c.getRadius()
          << "; Height = " << c.height;
   return output;    // разрешает конкатенацию вызовов
}
```

Рис. 20.2. Определения функций-элементов класса **Cylinder** (часть 7 из 9)

функций объектов тех классов, на которые ссылается **ptr**. Из вывода программы видно, что функции вызываются правильно для объекта **point**: выводится строка, начинающаяся с символов "**Point:** ", и значения площади и объема, равные **0.00**. Затем **ptr** присваивается адрес объекта **circle** и вызываются те же самые функции. Программа выдает правильный результат — выводится строка "**Circle:** ", вычисленная площадь круга и объем, равный **0.00**. В заключение **ptr** получает адрес объекта **cylinder** и производятся вызовы тех же самых функций. Вывод программы подтверждает правильность вызовов функций — выводится строка "**Cylinder:** " и значения площади поверхности и объема цилиндра. Все вызовы функций **printShapeName**, **area** и **volume** разрешаются при помощи механизма динамического связывания во время выполнения программы.

```
// FIG20_2.CPP
// Программа-тестер иерархии классов Point, Circle и Cylinder
#include <iostream.h>
#include <iomanip.h>
#include "shape.h"
#include "point1.h"
#include "circle1.h"
#include "cylindr1.h"

main()
{
   // создание геометрических фигур
   Point point(7, 11);
   Circle circle(3.5, 22, 8);
   Cylinder cylinder(10, 3.3, 10, 10);

   point.printShapeName();      // статическое связывание
   cout << point << endl;

   circle.printShapeName();     // статическое связывание
   cout << circle << endl;

   cylinder.printShapeName();   // статическое связывание
   cout << cylinder << "\n\n";

   cout << setiosflags(ios::showpoint) << setprecision(2);
   Shape *ptr;          // определение указателя на базовый класс

   // указатель на базовый класс ссылается на объект производного
   // класса Point
   ptr = &point;
   ptr->printShapeName();       // динамическое связывание
   cout << "x = " << point.getX()
        << "; y = " << point.getY()
        << "\nArea = " << ptr->area()
        << "\nVolume = " << ptr->volume() << "\n\n";

   // указатель на базовый класс ссылается на объект производного
   // класса Circle
   ptr = &circle;
   ptr->printShapeName();       // динамическое связывание
```

Рис. 20.2. Программа-тестер иерархии объектов **point**, **circle** и **cylinder** (часть 8 из 9)

```
   cout << "x = " << circle.getX()
        << "; y = " << circle.getY()
        << "\nArea = " << ptr->area()
        << "\nVolume = " << ptr->volume() << "\n\n";

   // указатель на базовый класс ссылается на объект производного
   // класса Cylinder
   ptr = &cylinder;
   ptr->printShapeName();        // динамическое связывание
   cout << "x = " << cylinder.getX()
        << "; y = " << cylinder.getY()
        << "\nArea = " << ptr->area()
        << "\nVolume = " << ptr->volume() << endl;

   return 0;
}
```

```
Point: [7, 11]
Circle: [22, 8]; Radius = 3.50
Cylinder: [10, 10]; Radius = 3.30; Height = 10.00

Point: x = 7.00; y = 11.00
Area = 0.00
Volume = 0.00

Circle: x = 22.00; y = 8.00
Area = 38.48
Volume = 0.00

Cylinder: x = 10.00; y = 10.00
Area = 275.77
Volume = 342.12
```

Рис. 20.2. Программа-тестер иерархии классов **point**, **circle** и **cylinder** (часть 9 из 9)

Резюме

- Благодаря виртуальным функциям и полиморфизму стало возможным разрабатывать системы, которые легко расширяются. Программы могут обрабатывать объекты, тип которых не был известен на момент их разработки.
- Виртуальные функции и полиморфизм позволяют отказаться от логики оператора **switch**. Механизм виртуальных функций автоматически обеспечивает необходимую логику поведения, и в результате исключаются ошибки, свойственные **switch**-логике. Программа-клиент, которая сама определяет типы объектов и их поведение, свидетельствует о *недостаточной проработке классов*.
- Виртуальная функция объявляется при помощи ключевого слова **virtual**, с которого начинается прототип функции в базовом классе.
- Производные классы могут вводить свои версии виртуальных функций базового класса, если им это нужно. Если они не делают этого, то получают в наследство функции базового класса.

- Если виртуальная функция вызывается при помощи имени объекта класса и операции-точки, ссылка разрешается во время компиляции (что называется статическим связыванием) и вызвана будет виртуальная функция, определенная в том классе (или унаследованная этим классом), к которому принадлежит данный объект.
- Во многих случаях бывает удобно определять классы, которые не предназначены для того, чтобы программист создавал объекты этих классов. Такие классы называются абстрактными. И так как они используются в наследовании в качестве базовых классов, то обычно их называют абстрактными базовыми классами. Объекты абстрактного класса нельзя создавать.
- Классы, объекты которых могут быть созданы, называются конкретными классами.
- Класс, имеющий виртуальные функции, может быть сделан абстрактным, если одну или несколько виртуальных функций объявить чистыми. Чистая виртуальная функция — виртуальная функция, в объявлении которой содержится инициализатор **=0**.
- Если класс выведен из класса с чистой виртуальной функцией и не определяет эту функцию, то функция остается чистой и в производном классе. Как следствие, производный класс тоже будет являться абстрактным классом.
- C++ поддерживает полиморфизм — способность объектов различных классов, связанных наследованием, реагировать по-разному на вызов одной и той же функции-элемента.
- Полиморфизм обеспечивается механизмом виртуальных функций.
- Когда виртуальная функция вызывается через посредство указателя на базовый класс, C++ выбирает замещающую функцию соответствующего производного класса, связанного с конкретным объектом, на который ссылается указатель.
- При использовании виртуальных функций и полиморфизма один и тот же вызов функции-элемента приводит к различным действиям в зависимости от типа объекта, получившего вызов.
- Хотя мы не можем создавать объекты абстрактных базовых классов, мы можем объявлять указатели на абстрактные базовые классы. Такие указатели могут затем использоваться для обеспечения полиморфного поведения объектов производных конкретных классов.
- В системы могут вводиться новые виды классов. Новые классы адаптируются к этим системам механизмом динамического (позднего) связывания. Для разрешения вызова виртуальной функции тип объекта не обязательно должен быть известен во время компиляции. Во время выполнения вызов виртуальной функции передается функции-элементу участвующего в вызове объекта.
- Динамическое связывание дает возможность независимым производителям программного обеспечения (ISV) распространять свои программы, не раскрывая своих секретов. Распространяемое ими программное обеспечение может состоять только из файлов заголовков и объектных файлов. Исходные тексты можно не распространять. Разработчики, получившие такой программный продукт, могут затем, используя меха-

низм наследования, выводить новые классы из полученных от ISV. Программное обеспечение, которое работает с поставляемыми ISV классами, будет работать и с классами, производными от них, вызывая (благодаря динамическому связыванию) замещающие виртуальные функции производных классов.

- При динамическом связывании во время выполнения программы вызов виртуальной функции-элемента перенаправляется функции из соответствующего класса. Для этой цели используется *таблица виртуальных функций*, называемая *vtable*, реализованная в виде массива, содержащего указатели на функции. Каждый класс, содержащий виртуальные функции, имеет таблицу *vtable*. Для каждой виртуальной функции класса в *vtable* включается указатель на версию виртуальной функции, используемую для объекта этого класса. Виртуальная функция, используемая в конкретном классе, может быть либо определена в нем, *либо унаследована*, непосредственно или косвенно, из вышестоящего в иерархии базового класса.
- Определенная в базовом классе виртуальная функция-элемент может быть переопределена в производном классе, но производные классы не должны делать это в обязательном порядке. Это значит, что производный класс может использовать виртуальную функцию-элемент из базового класса, что будет отражено в *vtable*.
- Каждый объект класса с виртуальными функциями содержит указатель на *vtable* своего класса. Этот указатель недоступен для программиста. Соответствующий указатель на функцию выбирается из *vtable* и разыменовывается; на этом завершается формирование во время выполнения вызова виртуальной функции. На просмотр *vtable* и разыменование указателя требуются незначительное время, обычно меньшее, чем в самом лучшем коде клиента.
- Если класс содержит виртуальные функции, то и деструктор базового класса следует объявлять виртуальным. Это приведет к тому, что деструкторы всех производных классов будут виртуальными, хотя их имена отличаются от имени деструктора базового класса. И теперь, если объект в иерархии разрушается при помощи операции **delete**, а указатель, ссылающийся на объект, является указателем на базовый класс, вызывается деструктор правильного производного класса.

Терминология

абстрактный базовый класс
абстрактный класс
альтернатива операторам **switch**
виртуальная функция
виртуальная функция базового класса
виртуальный деструктор
динамическое связывание
иерархия классов
ключевое слово virtual
конструктор производного класса
косвенный базовый класс
логика **switch**
наследование
непосредственный базовый класс
неявное преобразование указателей
объектно-ориентированное программирование
переопределенная виртуальная функция
повторное использование программного кода
позднее связывание
полиморфизм
преобразование объекта производного класса в объект

преобразование указателя производного класса в указатель базового класса
производный класс
раннее связывание
расширяемость
ссылка на абстрактный класс
ссылка на базовый класс
ссылка на производный класс
статическое связывание
указатель на абстрактный класс
указатель на базовый класс
указатель на производный класс
чисто виртуальная функция (=0)
явное преобразование указателей

Распространенные ошибки программирования

20.1. Определение в производном классе виртуальной функции, не совпадающей по типу возвращаемого значения или списку параметров с функцией в базовом классе, приводит к синтаксической ошибке.

20.2. Попытка создать объект абстрактного класса (т.е. содержащего одну или несколько чистых виртуальных функций) приводит к синтаксической ошибке.

20.3. Объявление конструктора виртуальной функцией. Конструктор не может быть виртуальным.

Хороший стиль программирования

20.1. Хотя функция производного класса является виртуальной, если на высших уровнях иерархии было сделано соответствующее объявление, многие программисты, для придания ясности программе, предпочитают использовать явное определение функции как виртуальной на всех уровнях иерархии.

20.2. Если класс имеет виртуальные функции, определяйте деструктор также виртуальным, даже если это и не требуется в данном конкретном случае. Производные от него классы могут иметь свои деструкторы, и тогда вызываться будут именно они.

Советы по повышению эффективности

20.1. Обеспечиваемый виртуальными функциями и динамическим связыванием полиморфизм довольно эффективен. Использование этих механизмов незначительно повлияет на эффективность системы.

20.2. Полиморфизм, обеспечиваемый виртуальными функциями и динамическим связыванием, можно рассматривать как конкурента оператора **switch**. И здесь стоит заметить, что обычно оптимизирующие компиляторы C++ для программ, использующих полиморфизм, генерируют столь же эффективный код, как и код на основе логики оператора **switch**.

Общие методические замечания

20.1. Интересным следствием использования виртуальных функций и полиморфизма является простота и ясность программ. В них становится меньше фрагментов с ветвлением и больше обычного линейного кода. Такие программы проще тестировать, отлаживать и сопровождать.

20.2. Если функция была объявлена как виртуальная, она остается виртуальной во всей иерархии наследования.

20.3. Если в производном классе виртуальная функция не определяется, то он просто наследует функцию своего непосредственного класса-предшественника.

20.4. Переопределяемая виртуальная функция должна иметь тот же возвращаемый тип и список параметров, что и виртуальная функция базового класса.

20.5. Если в классе, базовый класс которого содержит чистую виртуальную функцию, эта функция не определяется, то она остается чистой и в этом (производном) классе. Как следствие, производный класс сам является абстрактным классом.

20.6. Благодаря полиморфизму и виртуальным функциям программист может сосредоточить свое внимание на общих задачах, переложив частные вопросы на среду исполнения. Программист может управлять большим набором объектов, каждый из которых имеет свое специфическое поведение, не зная даже типов этих объектов.

20.7. Полиморфизм обеспечивает расширяемость системы: программный код, опирающийся на полиморфное поведение объектов, не зависит от типа объектов, которым передается вызов. Таким образом, без всяких изменений основной системы, в нее может быть введен новый тип объектов, который может реагировать на существующий набор сообщений. Программы не придется перекомпилировать, за исключением, возможно, некоторой части клиента, в которой создаются объекты нового типа.

20.8. В абстрактном классе определяется интерфейс всех узлов иерархии классов. Абстрактный класс содержит чистые виртуальные функции, которые будут определятся в производных классах. Благодаря полиморфизму этот интерфейс класса будет доступен всем функциям в иерархии.

Упражнения для самоконтроля

20.1. Вставьте пропущенные слова в следующих утверждениях:

a) Использование наследования и полиморфизма позволяет отказаться от логики _____.

b) Чистая виртуальная функция определяется символами _____ в конце прототипа, объявляемого в базовом классе.

c) Если класс содержит одну или несколько чистых виртуальных функций, он является _____.

d) Определение вызываемой функции во время компиляции называется _____ связыванием.

e) Определение вызываемой функции во время исполнения называется _____ связыванием.

Ответы на упражнения для самоконтроля

20.1. a) оператора **switch**. b) = 0. c) абстрактным базовым классом. d) статическим. e) динамическим.

Упражнения

20.2. Что такое виртуальные функции? Опишите, при каких обстоятельствах следует применять виртуальные функции.

20.3. Зная, что конструкторы не могут быть виртуальными, предложите вариант, как можно реализовать нечто подобное.

20.4. В упражнении 19.5 вы разработали иерархию класса **Shape** и определили классы иерархии. Измените иерархию так, чтобы класс **Shape** был абстрактным базовым классом и определял интерфейс классов иерархии. Выведите из **Shape** классы **TwoDimensionalShape** и **ThreeDimensionalShape** — эти классы также должны быть абстрактными. Для вывода типа и размеров объекта определите виртуальную функцию **print**. Также определите функции **area** и **volume** для вычисления соответствующих параметров объектов каждого конкретного класса в иерархии. Напишите программу для тестирования иерархии класса **Shape**.

20.5. Используя шаблон класса **List** из упражнения 17.10, создайте связанный список указателей на объекты класса **Shape** (т.е. указателей на любой объект в иерархии). Определите перегруженную операцию передачи в поток для класса **Shape**, который просто вызывает чистую виртуальную функцию **print**. Используя возможности вывода элементов связанного списка, пройдите по нему и выведите каждый элемент.

г л а в а

21

Потоки ввода/вывода в C++



Цели

- Понять принципы объектно-ориентированного потокового ввода/вывода C++.
- Научиться форматированию ввода и вывода.
- Изучить иерархию классов потоков ввода/вывода.
- Изучить организацию ввода/вывода объектов определяемого пользователем типа.
- Научиться создавать пользовательские манипуляторы потока.
- Научиться различать успешный или неудачный результат операций ввода/вывода.
- Освоить привязку выходного потока ко входному.

Содержание

21.1. Введение

21.2. Потоки

21.2.1. Заголовочные файлы библиотеки **iostream**

21.2.2. Классы и объекты потокового ввода/вывода

21.3. Потоковый вывод

21.3.1. Операция передачи в поток

21.3.2. Конкатенация операций передачи в поток и *извлечения из потока*

21.3.3. Вывод переменных типа **char ***

21.3.4. Функция-элемент **put** для вывода символов;
Конкатенация вызовов **put**

21.4. Потоковый ввод

21.4.1. Операция извлечения из потока

21.4.2. Функции-элементы **get** и **getline**

21.4.3. Другие функции-элементы класса **istream** (**peek**, **putback**, **ignore**)

21.4.4. Безопасный по типу ввод/вывод данных

21.5. Функции неформатируемого ввода/вывода *read*, *gcount* и *write*

21.6. Манипуляторы потоков

21.6.1. Манипуляторы установки основания целых чисел: **dec**, **oct**, **hex** и **setbase**

21.6.2. Установка точности представления чисел с плавающей точкой (**precision**, **setprecision**)

21.6.3. Ширина поля (**setw**, **width**)

21.6.4. Определяемые пользователем манипуляторы

21.7. Флаги форматирования потока

21.7.1. Флаги форматирования (**setf**, **unsetf**, **flags**)

21.7.2. Конечные нули и десятичные точки (**ios::showpoint**)

21.7.3. Выравнивание (**ios::left**, **ios::right**, **ios::internal**)

21.7.4. Заполнение (**fill**, **setfill**)

21.7.5. Флаги установки основания целых чисел (**ios::dec**, **ios::oct**, **ios::hex**, **ios::showbase**)

21.7.6. Числа с плавающей точкой; научная нотация (**ios::scientific**, **ios::fixed**)

21.7.7. Управление верхним/нижним регистрами (**ios::uppercase**)

21.7.8. Установка и сброс флагов форматирования (**flags**, **setiosflags**, **resetiosflags**)

21.8. Состояния ошибки потоков

21.9. Ввод/вывод определяемых пользователем типов

21.10. Привязка потока вывода к потоку ввода

Резюме • Распространенные ошибки программирования • Хороший стиль программирования • Советы по повышению эффективности • Общие методические замечания • Упражнения для самоконтроля • Ответы на упражнения для самоконтроля • Упражнения

21.1. Введение

Стандартные библиотеки C++ предоставляют вам широкий выбор возможностей ввода/вывода. В этой главе обсуждается ряд методов, достаточных для выполнения основных операций ввода/вывода, а остальные — даются в виде краткого обзора.

Многие из описанных здесь особенностей ввода/вывода являются объектно-ориентированными. Читателю, несомненно, будет интересно узнать, как эти возможности реализованы в языке. Заметим, что такой стиль операций ввода/вывода интенсивно использует другие особенности C++, такие, как ссылки и перегрузка функций и операций.

Как мы увидим, C++ обеспечивает *безопасный по типу* ввод/вывод. Каждая операция ввода/вывода выполняется по-разному в зависимости от типа обрабатываемых данных. Если для специфического типа данных определена функция ввода/вывода, то она автоматически вызывается для обработки этого типа. Если для некоего типа данных не найдена соответствующая функция, компилятор выдает сообщение об ошибке. Такая защита не пропускает через систему недопустимый тип данных (как это происходило в C; из-за этого недостатка в C происходили трудноуловимые и часто причудливые ошибки).

В дополнение к вводу/выводу предопределенных типов в C++ пользователь может вводить операции для ввода/вывода своих собственных типов. *Расширяемость языка* — одна из наиболее ценных особенностей C++.

Правило хорошего программирования 21.1

В программах на C++ используйте ввод/вывод только в стиле языка C++ и не применяйте функции ввода/вывода из библиотеки C, хотя они и доступны в C++.

Общее методическое замечание 21.1

Ввод/вывод в C++ безопасен в отношении использования различных типов.

Общее методическое замечание 21.2

C ++ предлагает единообразный интерфейс ввода/вывода для предопределенных типов и типов, определяемых пользователем. Благодаря такому общему подходу облегчается процесс разработки программ и, в частности, повторное использование программного обеспечения.

21.2. Потоки

В C++ ввод/вывод данных производится *потоками* байтов. Поток — это просто байтовая последовательность. При операциях ввода байты направляются от устройства (например, клавиатуры, дисковода, сетевой платы) в основную память. В операциях вывода поток байтов направляется из основной памяти на устройство (например, экран монитора, принтер, дисковод, сетевую плату).

Прикладные программы ассоциируют с этими байтами некоторый специфический смысл. Байты могут представлять ASCII-символы, сырые данные внутреннего формата, графические изображения, оцифрованную речь, цифровое видео или любой другой вид информации, с которой работает прикладная программа.

Работа системных механизмов ввода/вывода сводится к перемещению байтов данных от устройств в память и обратно, и поддержании при этом целостности и надежности всех данных. В процессах передачи данных нередко участвуют механические устройства типа вращающегося магнитного диска, ленты, клавиатуры. Время, которое уходит на обмен данными с такими устройствами, обычно значительно превышает время, которое тратит процессор на обработку этих данных. Поэтому операции ввода/вывода требуют тщательного планирования и настройки для того, чтобы обеспечивалась максимальная производительность системы. Эти вопросы подробно обсуждаются в книгах по операционным системам (см. De90).

В C++ имеются возможности ввода/вывода «высокого уровня» и «низкого уровня». Низкоуровневый (т.е. неформатируемый) ввод/вывод обычно состоит в передаче заданного числа байт от устройства в память или из памяти — устройству. В таких операциях единицей информации является байт. Операции низкого уровня способны передавать, и передавать быстро, большие объемы информации, но в не слишком удобной для человека форме.

Люди предпочитают пользоваться операциями ввода/вывода высокого уровня, т.е. форматируемым вводом/выводом, в котором байты группируются в элементы данных типа целых чисел, чисел с плавающей точкой, символов, строк и определяемых пользователем типов. Такой, ориентированный на тип данных, подход требуется в большинстве задач ввода/вывода, кроме обработки данных большого объема.

Совет по повышению эффективности 21.1

Применяйте неформатируемый ввод/вывод при обработке файлов большого объема.

21.2.1. Заголовочные файлы библиотеки iostream

Библиотека языка C++ **iostream** предоставляет программисту сотни разнообразных возможностей ввода/вывода. Части библиотечного интерфейса содержатся в нескольких файлах заголовков.

Большинство программ C++ должно включать файл заголовка **iostream.h**, который содержит основную информацию, необходимую для всех операций потокового ввода/вывода. В файле заголовка **iostream.h** содержатся объекты-потоки **cin**, **cout**, **cerr** и **clog**, которые, как мы вскоре увидим, ассоциируются со стандартным потоком ввода, стандартным потоком вывода и стандартным потоком ошибок. Здесь поддерживаются возможности форматируемого и неформатируемого ввода/вывода.

Файл заголовка **iomanip.h** содержит информацию, необходимую для управления форматируемым вводом/выводом с помощью так называемых *параметризованных манипуляторов потока.*

Заголовок **fstream.h** содержит важную информацию, нужную для управляемого пользователем файлового ввода/вывода.

В заголовочном файле **strstream.h** содержится информация, необходимая для выполнения *форматирования в памяти.* Эти операции напоминают опе-

рации при обработке файла, только выполняются они не над файлами, а над массивами символов.

Заголовок **stdiostream.h** должен включаться в программы, которые совмещают операции ввода/вывода языков C и C++. Во вновь создаваемых программах следует избегать ввода/вывода в стиле C; но люди, занимающиеся поддержкой существующих программ на языке C или переводящих C-программы на C++, найдут такую возможность полезной.

Каждая система программирования на C++ содержит дополнительные библиотеки ввода/вывода для работы со специфическими устройствами, например, ввода/вывода аудио- и видео-информации.

21.2.2. Классы и объекты потокового ввода/вывода

Библиотека **iostream** содержит большое количество классов, поддерживающих разнообразные операции ввода/вывода. Класс **istream** поддерживает операции потокового ввода. Класс **ostream** обеспечивает выполнение операций потокового вывода. Класс **iostream** поддерживает операции обоих типов: потокового ввода и потокового вывода.

Оба класса, **istream** и **ostream**, выводятся простым наследованием из базового класса **ios**. Класс **iostream** выводится путем сложного наследования от классов **istream** и **ostream**. Отношения наследования этих классов изображены на рис. 21.1.

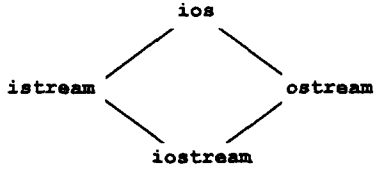


Рис. 21.1. Часть иерархии классов потоков ввода/вывода

Для операций ввода/вывода используется удобная нотация с применением знаков перегруженных операций. Для обозначения операции потокового вывода используется перегруженная операция левого сдвига (<<), которая называется операцией *передачи в поток*. Аналогично, для обозначения операции потокового ввода используется перегруженная операция правого сдвига (>>), которая называется операцией *извлечения из потока*. Эти операции могут применяться со стандартными объектами потоков **cin**, **cout**, **cerr**, **clog** и с объектами потоков, определяемыми пользователем.

Объект **cin** является объектом класса **istream**, ассоциированным со стандартным устройством ввода данных; обычно это клавиатура. В следующем операторе показана операция извлечения из потока; значение, полученное от объекта **cin**, присваивается переменной **grade** целого типа:

```
cin >> grade;
```

Заметим здесь, что операция извлечения из потока достаточно «сообразительна» и «знает», с каким типом данных имеет дело. Если переменная **grade** была должным образом объявлена, то тип переменной, участвующей в операции извлечения из потока, специально указывать не нужно (как это требуется, например, в случае ввода/вывода в языке C).

Объект **cout** класса **ostream** ассоциирован со стандартным устройством вывода; обычно это монитор. Операция передачи в поток, использованная в следующем операторе, выводит значение целой переменной **grade** (из памяти) на стандартное устройство вывода:

```
cout << grade;
```

Заметим, что операция передачи в поток также достаточно «умна» и «знает» тип переменной **grade** (которая должна быть должным образом объявлена), так что никакая дополнительная информация о типе для операции помещения в поток не требуется.

Объект **cerr** класса **ostream** ассоциируется со стандартным устройством ошибок с небуферизованным выводом. Последнее означает, что результат каждой операции передачи в поток **cerr** выводится немедленно, а не помещается в буфер; это необходимо для оперативного оповещения пользователя о возникшей проблеме.

Объект **clog** принадлежит классу **ostream** и ассоциируется со стандартным устройством ошибки. Вывод на **clog** буферизован.

При работе с файлами в C++ класс **ifstream** используется для выполнения файловых операций ввода, класс **ofstream** — для операций файлового вывода и класс **fstream** — для обоих видов операций. Класс **ifstream** является потомком класса **istream**, класс **ofstream** — потомок **ostream** и класс **fstream** — производный от класса **iostream**. Связи наследования этих классов ввода/вывода графически представлены на рис. 21.2. Полная иерархия классов потокового ввода/вывода, поддерживаемая в большинстве систем, включает в себя намного больше классов, однако классы, показанные здесь, обеспечивают достаточно широкие возможности и для большинства программистов их будет достаточно. За подробной информацией по работе с файлами обращайтесь к описанию библиотеки классов вашей системы программирования на C++.

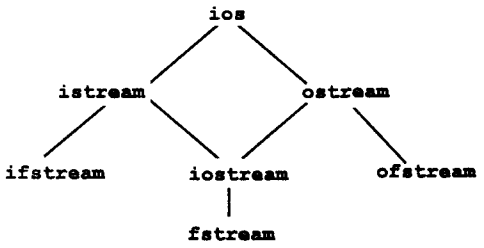


Рис. 21.2. Часть иерархии классов потокового ввода/вывода с основными классами файлового ввода/вывода

21.3. Потоковый вывод

Класс **ostream** в C++ обеспечивает выполнение форматируемого и неформатируемого вывода. Его возможности включают в себя: вывод стандартных типов данных при помощи операции передачи в поток; вывод символов при помощи функции-элемента **put**; неформатируемый вывод, осуществляемый функцией-элементом **write** (раздел 21.5); вывод целых чисел в десятичном, восьмеричном и шестнадцатеричном форматах (раздел 21.6.1); вывод чисел с плавающей точкой с различной точностью (раздел 21.6.2), с принудительной

десятичной точкой (раздел 21.7.2), в экспоненциальном (научном) представлении и с фиксированной точкой (раздел 21.7.6); вывод данных с выравниванием в полях заданной ширины (раздел 21.7.3); вывод данных с символами-заполнителями (раздел 21.7.4) и использование символов верхнего регистра в научной и шестнадцатеричной записи (раздел 21.7.7).

21.3.1. Операция передачи в поток

Вывод данных в поток выполняется при помощи операции передачи в поток, а именно, перегруженной операции левого сдвига <<. Перегруженная операция << может выводить данные предопределенных типов, строки символов и значения указателей. В разделе 21.9 мы узнаем, как перегрузить операцию <<, чтобы ее можно было использовать с определяемыми пользователем типами данных. На рис. 21.3 показан пример вывода строки символов.

```
// fig21_03.cpp
// Вывод строки с использованием операции передачи в поток.

#include <iostream.h>

main()
{
   cout << "Welcome to C++!\n";

   return 0;
}
```

```
Welcome to C++!
```

Рис. 21.3. Вывод строки с использованием операции передачи в поток

В указанном примере используется один оператор с операцией передачи в поток для вывода строки. Можно написать несколько операторов вывода в поток, как на рис. 21.4. При выполнении эта программа выводит ту же самую строку, что и предыдущая.

```
// fig21_04.cpp
// Вывод строки с использованием двух операций передачи в поток.

#include <iostream.h>

main()
{
   cout << "Welcome to ";
   cout << "C++!\n";

   return 0;
}
```

```
Welcome to C++!
```

Рис. 21.4. Вывод строки с использованием двух операций передачи в поток

Тот же самый эффект, что дает esc-код \n («новая строка»), может быть получен при помощи *манипулятора потока* **endl** («конец строки»), как показано на рис. 21.5. Манипулятор потока **endl** выдает символ «новая строка» и, кроме того, сбрасывает данные из буфера вывода (т.е. принудительно выводит данные из буфера, даже если он не заполнен). Буфер вывода также может быть очищен при помощи манипулятора

```
cout << flush;
```

Манипуляторы потоков подробно обсуждаются в разделе 21.6.
Можно выводить и выражения, как показано на рис. 21.6.

```
// fig21_05.cpp
// Использование манипулятора endl.
#include <iostream.h>

main()
{
   cout << "Welcome to ";
   cout << "C++!";
   cout << endl;  // потоковый манипулятор "конец строки"

   return 0;
}
```

```
Welcome to C++!
```

Рис. 21.5. Применение манипулятора **endl**

```
// fig21_06.cpp
// Вывод значения выражения.
#include <iostream.h>

main()
{
   cout << "47 plus 53 is ";
   cout << 47 + 53;       // выражение
   cout << endl;

   return 0;
}
```

```
47 plus 53 is 100
```

Рис. 21.6. Вывод значения выражения

Хороший стиль программирования 21.2

При выводе выражений заключайте их в скобки, чтобы избежать конфликтов, связанных со старшинством операции << и операций в выражении.

21.3.2. Конкатенация операций помещения в поток и извлечения из потока

Перегруженные операции << и >> могут использоваться *в форме конкатенации*, как показано на рис. 21.7.

```
// fig21_07.cpp
// Конкатенация операций передачи в поток.
#include <iostream.h>

main()
{
   cout << "47 plus 53 is " << 47 + 53 << endl;

   return 0;
}
```

```
47 plus 53 is 100
```

Рис. 21.7. Конкатенация операций передачи в поток

Операции передачи в поток из примера на рис. 21.7 выполняются таком в порядке, как если бы они были записаны следующим образом:

```
(((cout << "47 plus 53 is ") << 47 + 53) << endl);
```

Конкатенация возможна потому, что перегруженная операция << возвращает ссылку на свой левый операнд, т.е. **cout**. Таким образом, крайнее левое выражение в скобках будет выполнено первым и

```
(cout << "47 plus 53 is ")
```

выведет заданную строку символов и возвратит ссылку на **cout**. В результате следующее по порядку вычисления выражение в скобках будет преобразовано к виду

```
(cout << 47 + 53)
```

затем будет выведено число **100** и возвращена ссылка на **cout**. Последнее выражение в скобках примет вид

```
cout << endl
```

и в результате его выполнения в поток вывода будет помещен символ новой строки, очищен буфер потока **cout** и возвращена ссылка на **cout**. Последняя ссылка на **cout** использована не будет.

21.3.3. Вывод переменных типа char *

При вводе/выводе в C необходимо было сообщать соответствующую информацию о типе данных. В C++ тип данных определяется автоматически, что является очень удобным нововведением. Но иногда этот автоматизм «встает поперек дороги». Рассмотрим пример. Как мы знаем, символьная строка имеет тип **char ***. Предположим, что мы хотим вывести значение такого указателя, т.е. адрес памяти первого символа строки. Но перегруженная операция << выводит данные типа **char *** как строку символов с завершающим нуль-символом. Решение состоит в том, чтобы привести указатель к типу **void *** (и так нужно поступать с указателем любого типа, когда нужно вывести адрес, на который ссылается указатель). На рис. 21.8 переменная типа **char *** выводится в виде адреса и строки символов, расположенной по этому адресу. Обратите внимание, что адрес выводится в виде шестнадцатеричного (по основанию 16) числа. Шестнадцатеричные числа в C++ начинаются с символов **0x** или **0X**. Мы поговорим подробнее о различных системах представления чисел в разделах 21.6.1, 21.7.4, 21.7.5 и 21.7.7.

```
// fig21_08.cpp
// Вывод значения адреса, содержащегося в переменной-указателе
// на тип char
#include <iostream.h>

main()
{
   char *string = "test";

   cout << "Value of string is: " << string
        << "\nValue of (void *)string is: "
        << (void *)string << endl;

   return 0;
}
```

```
Value of string is: test
Value of (void *)string is: 0x00aa
```

Рис. 21.8. Вывод значения адреса, содержащегося в переменной-указателе на тип **char**

21.3.4. Функция-элемент put для вывода символов; конкатенация вызовов put

Символ можно вывести с помощью функции-элемента **put**, как в следующем примере,

```
cout.put('A');
```

в котором на экран выводится символ **A**. Вызовы **put** могут конкатенироваться:

```
cout.put('A').put('\n');
```

При этом сначала выводится символ **A**, а за ним — символ новой строки. Функции **put** можно передавать в качестве аргумента выражение, оценивающееся как ASCII-код символа, например, **cout.put(65)**, и в результате на экран также будет выведен символ **A**.

21.4. Потоковый ввод

Теперь давайте рассмотрим потоки ввода. Для получения данных из потока используется перегруженная операция правого сдвига >>, называемая *операцией извлечения* (из потока). Эта операция по умолчанию пропускает в потоке ввода *пробельные символы* (такие, как пробел, символы табуляции и новой строки). Позже мы узнаем, как можно изменить обработку этих символов. Операция извлечения из потока возвращает нуль (false), когда в потоке встречается признак конца файла. Каждый поток имеет набор *битов состояния*, с помощью которых можно управлять поведением потока (т.е. форматированием, состояниями ошибок и т.д.). При вводе неправильного типа операция извлечения из потока устанавливает бит потока **failbit**, в случае завершения операции с ошибкой — бит **badbit**. Мы скоро узнаем, как можно опросить состояние потока после завершения операций ввода/вывода. В разделах 21.7 и 21.8 биты состояния потока обсуждаются подробно.

21.4.1. Операция извлечения из потока

Если вам нужно ввести два целых числа, используйте объект-поток **cin** и перегруженную операцию >> извлечения из потока, как показано на рис. 21.9.

```
// fig21_09.cpp
// Вычисление суммы двух чисел,
// введенных с клавиатуры и полученных из потока cin
// с помощью операции извлечения из потока.
#include <iostream.h>

main()
{
   int x, y;

   cout << "Enter two integers: ";
   cin >> x >> y;
   cout << "Sum of " << x << " and " << y << " is: "
        << x + y << endl;

   return 0;
}
```

```
Enter two integers: 30 92
Sum of 30 and 92 is: 122
```

Рис. 21.9. Вычисление суммы двух чисел, введенных с клавиатуры и полученных из потока **cin** с помощью операции извлечения

Относительно высокий приоритет операций >> и << может порождать некоторые проблемы. Например, если в программе на рис. 21.10 вы не заключите в круглые скобки условное выражение, то программа будет компилирована неверно. Читателю стоит проверить это утверждение.

```
// fig21_10.cpp
// Пример конфликта, связанного с приоритетом операций.
// Условное выражение необходимо заключить в скобки.
#include <iostream.h>

main ()
{
   int x, y;

   cout << "Enter two integers: ";
   cin >> x >> y;
   cout << x << (x == y ? " is" : " is not")
        << " equal to " << y << endl;

   return 0;
}
```

```
Enter two integers: 7 5
7 is not equal to 5

Enter two integers: 8 8
8 is equal to 8
```

Рис. 21.10. Как избежать конфликта старшинства операций между операцией передачи в поток и условным выражением

Распространенная ошибка программирования 21.1

Попытка чтения данных из потока **ostream** (или любого другого потока вывода).

Распространенная ошибка программирования 21.2

Попытка записи данных в поток **istream** (или любой другой поток ввода).

Распространенная ошибка программирования 21.3

В операторе ввода/вывода из потока опускают скобки, и в результате из-за относительно высокого приоритета << и >> операции выполняются в неправильном порядке.

Один из популярных способов ввода ряда значений — это размещение операции извлечения из потока в заголовке цикла **while**, т.к. операция извлечения возвращает значение false (нуль), когда в потоке встречается признак конца файла. Рассмотрим программу на рис 21.11, которая определяет максимальное значение среди введенных чисел. Имейте в виду, что число вводимых данных не известно заранее и, чтобы обозначить конец ввода, пользователь должен ввести комбинацию клавиш для конца файла. Условие цикла **while (cin >> grade)** становится равным нулю (принимает значение false), когда пользователь вводит конец файла.

```
// fig21_11.cpp
// Операция извлечения из потока возвращает
// значение false при вводе конца файла.
#include <iostream.h>

main()
{
   int grade, highestGrade = -1;

   cout << "Enter grade (enter end-of-file to end): ";
   while (cin >> grade) {
      if (grade > highestGrade)
         highestGrade = grade;

      cout << "Enter grade (enter end-of-file to end): ";
   }

   cout << "\nHighest grade is: " << highestGrade
        << endl;
   return 0;
}
```

```
Enter grade (enter end-of-file to end): 67
Enter grade (enter end-of-file to end): 87
Enter grade (enter end-of-file to end): 73
Enter grade (enter end-of-file to end): 95
Enter grade (enter end-of-file to end): 34
Enter grade (enter end-of-file to end): 99
Enter grade (enter end-of-file to end): ^Z

Highest grade is: 99
```

Рис. 21.11. Операция извлечения из потока возвращает значение false при вводе признака конца файла

21.4.2. Функции-элементы get и getline

Функция-элемент **get** без аргументов вводит один символ из указанного потока ввода (даже если этот символ — пробельный) и возвращает его в качестве своего значения. Эта версия функции **get** возвращает макрос **EOF**, когда в потоке ввода встречается конец файла. Макрос **EOF** обычно имеет значение **-1**, чтобы это значение можно было отличить от ASCII-символа (значение **EOF** может различаться в разных системах).

На рис. 21.12 показан пример использования функций-элементов **eof** и **get** объекта-потока **cin** и функции-элемента **put** объекта-потока **cout**. Программа сначала выводит значение вызова **cin.eof()**, т.е. **0** (false) для того, чтобы показать, что конец файла в **cin** еще не встретился. Затем пользователь вводит строку текста, завершая ее концом файла (**<ctrl>-z** на IBM PC и совместимых системах,**<return>**, **<ctrl>-d** для UNIX и Macintosh). Программа читает каждый символ и выводит его в поток **cout**, вызывая функцию-элемент **put**. При вводе конца файла цикл **while** завершается и возвращаемое значение **cin.eof()** — а теперь это **1** (true) — завершает вывод программы, демонстрируя тем самым, что для **cin** установлен признак конца файла. Обратите внимание, что в этой программе используется версия функции-элемента **get** класса **istream**, которая не принимает аргументов и возвращает введенный символ.

```
// fig21_12.cpp
// Использование функций-элементов get, put и eof.
#include <iostream.h>

main()
{
   int c;

   cout << "Before input, cin.eof() is "
        << cin.eof() << '\n'
        << "Enter a sentence followed by end-of-file:\n";

   while ((c = cin.get()) != EOF)
      cout.put(c);

   cout << "\nAfter input, cin.eof() is "
        << cin.eof() << endl;

   return 0;
}
```

```
Before input, cin.eof() is 0
Enter a sentence followed by end-of-file:
Testing the get and put member functions^Z
Testing the get and put member functions
After input, cin.eof() is 1
```

Рис. 21.12. Использование функций-элементов **get**, **put** и **eof**

Функция-элемент **get** с символьным аргументом извлекает из потока ввода следующий символ (включая и пробельные символы). Эта версия функции **get** возвращает значение false при появлении конца файла; в противном слу-

чае эта версия возвращает ссылку на объект класса **istream**, для которого функция **get** вызывалась.

Третья версия функции-элемента **get** имеет три аргумента — символьный массив, размер массива и символ-ограничитель (по умолчанию это '**\n**'). Эта версия считывает символы из входного потока следующим образом: вводится на один символ меньше, чем указанный во втором аргументе размер массива, после чего ввод прекращается, либо ввод прекращается досрочно, как только встретится символ-ограничитель. В конец строки, считанной в символьный массив, дописывается нуль-символ. Символ-ограничитель не записывается в символьный массив, а остается в потоке ввода. На рис. 21.13 приводится пример, сравнивающий ввод из потока **cin** при помощи операции извлечения из потока и функции **cin.get**.

```
// fig21_13.cpp
// Сравнение ввода строки символов с использованием
// cin и cin.get.
#include <iostream.h>

const int SIZE = 80;

main()
{
   char buffer1[SIZE], buffer2[SIZE];

   cout << "Enter a sentence:\n";
   cin >> buffer1;
   cout << "\nThe string read with cin was:\n"
        << buffer1 << "\n\n";

   cin.get(buffer2, SIZE);
   cout << "The string read with cin.get was:\n"
        << buffer2 << endl;

   return 0;
}
```

```
Enter a sentence:
Contrasting string input with cin and cin.get

The string read with cin was:
Contrasting

The string read with cin.get was:
 string input with cin and cin.get
```

Рис. 21.13. Ввод строки символов из потока **cin** при помощи операции извлечения из потока и функции **cin.get**

Функция-элемент **getline** работает подобно третьей версии функции-элемента **get**, также добавляя нуль-символ в конец считанной в символьный массив строки. Функция **getline** удаляет символ-ограничитель из потока, но не сохраняет его в массиве. Программа на рис. 21.14 демонстрирует использование функции-элемента **getline** при вводе строки текста.

```
// fig21_14.cpp
// Ввод символов при помощи функции-элемента getline.

#include <iostream.h>

const SIZE = 80;

main()
{
   char buffer[SIZE];

   cout << "Enter a sentence:\n";
   cin.getline(buffer, SIZE);

   cout << "\nThe sentence entered is:\n"
        << buffer << endl;

   return 0;
}
```

```
Enter a sentence:
Using the getline member function

The sentence entered is:
Using the getline member function
```

Рис. 21.14. Ввод символов при помощи функции-элемента **getline**

21.4.3. Другие функции-элементы класса istream (peek, putback, ignore)

Функция-элемент **ignore** пропускает указанное число символов (по умолчанию — один символ) или завершается, если встретит указанный символ-ограничитель (по умолчанию им является макрос **EOF**).

Функция-элемент **putback** возвращает последний символ, полученный функцией **get** из потока ввода, обратно в поток. Эта функция удобна для использования в прикладных программах, которые просматривают входной поток в поисках фрагмента, начинающегося со специфического символа. Когда такой символ обнаруживается, приложение возвращает его обратно в поток, чтобы нужный фрагмент данных затем был считан целиком.

Функция-элемент **peek** возвращает следующий символ из потока ввода, но не удаляет его из потока.

21.4.4. Безопасный по типу ввод/вывод данных

C++ реализует *безопасный по типу* ввод/вывод данных. Операции << и >> перегружены для работы со специфическими типами данных. При обработке недопустимого типа устанавливаются различные флаги ошибок, которые пользователь может проверить, чтобы определить, завершилась ли операция ввода/вывода успешно или неудачно. Таким образом программа «держит все под контролем». Мы обсудим эти флаги ошибок в разделе 21.8.

21.5. Функции неформатируемого ввода/вывода read, gcount и write

Неформатируемый ввод/вывод выполняется функциями-элементами **read** и **write**. Каждая из этих функций вводит или выводит некоторое число байт в массив или из массива в памяти. Эти блоки байт никак не форматируются. Они просто вводятся или выводятся в сыром виде. Например, в результате выполнения операторов

```
char buffer[] = "HAPPY BIRTHDAY";
cout.write(buffer, 10);
```

первые 10 байт массива **buffer** будут выведены на экран. Так как символьная строка представляется адресом ее первого символа, то в результате следующего вызова

```
cout.write( "ABCDEFGHI JKLMNOPQRSTUVWXYZ", 10);
```

будут выведены первые 10 символов алфавита.

Функция-элемент **read** вводит в символьный массив определенное количество символов. Если прочитано меньшее количество символов, чем требовалось, устанавливается бит **failbit**. Мы скоро узнаем, как определить состояние этого бита (см. раздел 21.8).

В программе на рис. 21.15 показан пример применения функций-элементов **read** и **gcount** класса **istream** и функции-элемента **write** класса **ostream**. Программа вводит 20 символов (из более длинной строки ввода) в символьный массив **buffer**, используя функцию **read**, определяет количество реально введенных символов при помощи функции **gcount** и выводит символы из **buffer** функцией **write**.

```
// fig21_15.cpp
// Неформатируемый ввод/вывод с использованием
// функций-элементов read, gcount и write.
#include <iostream.h>

const int SIZE = 80;

main()
{
   char buffer[SIZE];

   cout << "Enter a sentence:\n";
   cin.read(buffer, 20);
   cout << "\nThe sentence entered was:\n";
   cout.write(buffer, cin.gcount());
   cout << endl;

   return 0;
}
```

```
Enter a sentence:
Using the read, write, and gcount member functions

The sentence entered was:
Using the read, writ
```

Рис. 21.15. Неформатируемый ввод/вывод с использованием функций-элементов **read**, **gcount** и **write**

21.6. Манипуляторы потоков

В C++ имеются различные *манипуляторы потоков*, которые выполняют форматирование данных. Манипуляторы потоков выполняют следующие задачи: установка ширины поля, задание точности представления, установка и сброс флагов форматирования, задание символа-заполнителя полей, очистка потоков, вставка в поток вывода символа новой строки и очистка потока, вставка нуль-символа в поток вывода и пропуск пробельных символов во входном потоке. Эти моменты описываются в следующих разделах.

21.6.1. Манипуляторы установки основания целых чисел: dec, oct, hex и setbase

Целые числа обычно представляются в десятичной системе счисления (по основанию 10). Для того, чтобы изменить основание системы счисления, в которой целые числа будут интерпретироваться в потоке, используют манипулятор **hex** — для шестнадцатеричной системы (по основанию 16), манипулятор **oct** — для восьмеричной системы (по основанию 8). Манипулятор потока **dec** возвращает поток к десятичной системе счисления.

Основание системы счисления в потоке также может быть изменено манипулятором потока **setbase**, который имеет один целочисленный аргумент, принимающий значения **10**, **8** или **16** для соответствующей системы счисления. Поскольку манипулятор **setbase** имеет аргумент, он называется *параметризованным манипулятором потока*. При использовании **setbase** или любого другого параметризованного манипулятора необходимо включить в программу файл заголовка **iomanip.h**. Установленное в потоке основание остается в силе, пока не будет явно изменено. В программе на рис. 21.16 показано применение манипуляторов **hex**, **oct**, **dec** и **setbase**.

```
// fig21_16.cpp
// Использование потоковых манипуляторов hex, oct, dec и setbase
#include <iostream.h>
#include <iomanip.h>

main()
{
   int n;

   cout << "Enter a decimal number: ";
   cin>>n;

   cout << n << " in hexadecimal is: "
        << hex << n << '\n'
        << dec << n << " in octal is:
        << oct << n << '\n'
        << setbase(10) << n << " in decimal is: "
        << n << endl;

return 0;
}
```

Рис. 21.16. Применение потоковых манипуляторов **hex**, **oct**, **dec** и **setbase** (часть 1 из 2)

```
Enter a decimal number: 20
20 in hexadecimal is: 14
20 in octal is: 24
20 in decimal is: 20
```

Рис. 21.16. Применение потоковых манипуляторов **hex**, **oct**, **dec** и **setbase** (часть 2 из 2)

21.6.2. Установка точности представления чисел с плавающей точкой (precision, setprecision)

Мы можем управлять *точностью* представления чисел с плавающей точкой, а именно — числом цифр справа от десятичной точки[1], используя манипулятор потока **setprecision** или функцию-элемент **precision**. Любой из этих вызовов устанавливает точность для всех последующих операций вывода, пока не будет сделан следующий вызов. Функция-элемент **precision** без аргумента возвращает текущую установку точности. В программе на рис. 21.17 используются оба способа — и функция-элемент **precision** и манипулятор **setprecision** — для вывода таблицы квадратного корня из двойки с точностью, изменяющейся от **0** до **9**. Обратите внимание, что точность **0** имеет специфическое назначение. Это значение аргумента восстанавливает *заданную по умолчанию точность*, равную **6**.

21.6.3. Ширина поля (setw, width)

Функция-элемент **width** устанавливает ширину поля и возвращает предыдущее значение этой величины. В случае, если обрабатываемые значения занимают места меньше, чем ширина поля, в оставшихся позициях выводится *символ-заполнитель*. Если значение требует для своего размещения места больше, чем отведенная ширина поля, — усечение не производится, значение выводится полностью. Установленное значение ширины применяется только для последующей операции передачи или извлечения, после чего ширина поля устанавливается равной **0**, т.е. выводимые значения будут занимать поля ширины, необходимой для вывода данного значения. Функция **width** без аргумента возвращает текущее значение ширины поля.

Распространенная ошибка программирования 21.4

Если установленное значение ширины поля меньше, чем необходимо для вывода, выводимая величина будет помещена в поле необходимой ширины, но при этом результат вывода будет неудобочитаем или будет восприниматься неверно.

На рис. 21.18 показано использование функции-элемента **width** при вводе и выводе. Обратите внимание, что при вводе максимально считывается на один символ меньше, чем ширина поля, — оставляется место для нуль-символа, добавляемого в конец введенной строки. Помните, что операция извлечения из потока завершается при появлении в потоке пробельного символа (если он не ведущий). Для задания ширины поля можно также использовать манипулятор потока **setw**.

1 Если установлен флаг форматирования **ios::scientific** или **ios::fixed**. Если ни один из этих флагов не установлен, «точность» задает *общее число* значащих цифр. Пример из рис. 21.17 на самом деле будет выводить для каждого числа на одну цифру меньше, чем показано на рисунке. — *Прим. ред.*

```
// fig21_17.cpp
// Управление точностью представления чисел с плавающей точкой
#include <iostream.h>
#include <iomanip.h>
#include <math.h>

main()
{
   double root2 = sqrt(2.0);

   cout << "Square root of 2 with precisions 0-9.\n"
        << "Precision set by the "
        << "precision member function:\n";

   for (int places = 0; places <= 9; places++) {
      cout.precision(places);
      cout << root2 << endl;
   }

   cout << "\nPrecision set by the "
        << "setprecision manipulator:\n";

   for (places = 0; places <= 9; places++)
      cout << setprecision(places) << root2 << endl;

   return 0;
}
```

```
Square root of 2 with precisions 0-9.
Precision set by the precision member function:
1.414214
1.4
1.41
1.414
1.4142
1.41421
1.414214
1.4142136
1.41421356
1.414213562

Precision set by the setprecision manipulator:
1.414214
1.4
1.41
1.414
1.4142
1.41421
1.414214
1.4142136
1.41421356
1.414213562
```

Рис. 21.17. Управление точностью представления чисел с плавающей точкой

```
// fig21_18.cpp
// Использование функции-элемента width

#include < iostream. h>

main()
{
   int w = 4;
   char string[10];

   cout << "Enter a sentence:\n";
   cin.width(5);

   while (cin >> string) {
      cout.width(w++);
      cout << string << '\n';
      cin.width(5);
   }

   return 0;
}
```

```
Enter a sentence:
This is a test of the width member function^Z
This
   is
     a
   test
      of
      the
      widt
          h
        memb
           er
          func
           tion
```

Рис. 21.18. Использование функции-элемента **width**

21.6.4. Определяемые пользователем манипуляторы

Пользователи могут создавать свои собственные манипуляторы потока. На рис. 21.19 показано создание и проверка новых манипуляторов потока **bell** (звонок), **ret** (возврат каретки), **tab** и **endLine**. Пользователи могут создавать собственные параметризованные манипуляторы потока. О том, как это сделать, смотрите в руководствах по вашему компилятору.

21.7. Флаги форматирования потока

Потоки C++ имеют набор различных *флагов форматирования*, которые определяют преобразования данных, выполняемые при операциях потокового ввода/вывода. Для работы с флагами используются функции-элементы **setf**, **unsetf** и **flags**.

```
// fig21_19.cpp
// Пример создания и использования определенных пользователем
// непараметризованных манипуляторов

#include <iostream.h>

// манипулятор bell (используется escape-код \a)
ostream& bell(ostream& output)
{
   return output << '\a';
}

// манипулятор ret (используется escape-код \r)
ostream& ret(ostream& output)
{
   return output << '\r';
}

// манипулятор tab (используется escape-код \t)
ostream& tab(ostream& output)
{
   return output << '\t';
}

// манипулятор endLine (используется escape-код \n
// и функция-элемент flush)
ostream& endLine(ostream& output)
{
   return output << '\n' << flush;
}

main()
{
   cout << "Testing the tab manipulator:\n"
        << 'a' << tab << 'b' << tab << 'c' << endLine
        << "Testing the ret and bell manipulators:\n"
        << ".........";

   for (int i = 1; i <= 100; i++)
      cout << bell;

   cout << ret << "-----" << endLine;

   return 0;
}
```

```
Testing the tab manipulator:
a       b       c
Testing the ret and bell manipulators:
-----.....
```

Рис. 21.19. Создание и проверка определенных пользователем непараметризованных манипуляторов

21.7.1. Флаги форматирования

Флаги форматирования, показанные на рис. 21.20, определены в классе **ios** как константы перечислимого типа и будут обсуждаться в следующих нескольких разделах.

```
ios::skipws

ios::left
ios::right
ios::internal

ios::dec
ios::oct
ios::hex

ios::showbase
ios::showpoint
ios::uppercase
ios::showpos

ios::scientific
ios::fixed

ios::unitbuf
ios::stdio
```

Рис. 21.20. Флаги форматирования

Этими флагами можно управлять с помощью функций-элементов **flags**, **setf** и **unsetf**, но многие программисты предпочитают использовать манипуляторы потока (см. раздел 10.7.8). Программист может использовать операцию поразрядного ИЛИ |, чтобы объединить различные опции в одном числе типа **long** (см. рис. 10.23). При вызове функции-элемента **flags** с таким набором битов устанавливаются новые опции форматирования для данного потока и возвращается значение, содержащее предыдущее состояние флагов. Возвращаемое значение обычно сохраняется для того, чтобы потом вызвать **flags** с этим значением и восстановить предыдущие опции потока.

Функция **flags** устанавливает значения всех флагов. Функция **setf** с одним аргументом, с другой стороны, специфицирует один или большее количество флагов, объединяемых операцией ИЛИ с существующими значениями флагов, формируя новое состояние форматирования.

Параметризованный манипулятор потока **setiosflags** выполняет те же самые функции, что и функция-элемент **setf**. Потоковый манипулятор **resetiosflags** выполняет те же самые функции, что и функция-элемент **unsetf**. Для того, чтобы использовать любой из этих потоковых манипуляторов, не забудьте включить в программу строку **#include <iomanip.h>**.

Установленный флаг **skipws** указывает, что операция >> должна пропускать пробельные символы в потоке ввода. Этот режим является режимом по умолчанию. Чтобы изменить эту установку, используйте вызов **unsetf(ios::skipws)**. Для задания режима пропуска пробельных символов также может использоваться манипулятор потока **ws**.

21.7.2. Конечные нули и десятичные точки (ios::showpoint)

При установленном флаге **showpoint** числа с плавающей точкой выводятся с десятичной точкой и конечными нулями. Число **79.0** будет выведено в виде **79**, если флаг **showpoint** сброшен, и как **79.000000** (или с количеством конечных нулей, соответствующим установленной точности), когда он установлен.

В программе на рис. 21.21 показано использование функции-элемента **setf** для установки флага **showpoint**, контролирующего вывод нулей в младших разрядах и десятичной точки у вещественных чисел.

```
// fig21_21.cpp
// Управление выводом нулей в младших разрядах
// и десятичной точки
// у чисел с плавающей точкой

#include <iostream.h>
#include <iomanip.h>
#include <math.h>

main()
{
   cout << "cout prints 9.9900 as: " << 9.9900
        << "\ncout prints 9.9000 as: " << 9.9000
        << "\ncout prints 9.0000 as: " << 9.0000
        << "\n\nAfter setting the ios::showpoint flag\n";
   cout.setf(ios::showpoint);

   cout << "cout prints 9.9900 as: " << 9.9900
        << "\ncout prints 9.9000 as: " << 9.9000
        << "\ncout prints 9.0000 as: " << 9.0000 << endl;

   return 0;
}
```

```
cout prints 9.9900 as: 9.99
cout prints 9.9000 as: 9.9
cout prints 9.0000 as: 9

After setting the ios::showpoint flag
cout prints 9.9900 as: 9.990000
cout prints 9.9000 as: 9.900000
cout prints 9.0000 as: 9.000000
```

Рис. 21.21. Управление выводом нулей в младших разрядах и десятичной точки для чисел с плавающей точкой

21.7.3. Выравнивание (ios::left, ios::right, ios::internal)

Флаги **left** и **right** позволяют выводить данные с левым выравниванием в поле и заполняющими символами в правой части поля, или с правым выравниванием и заполняющими символами слева соответственно. Символ, который нужно использовать для заполнения, определяется функцией-элементом **fill** или параметризованным манипулятором потока **setfill** (см. раздел 21.7.4).

На рис. 21.22 показан пример использования манипуляторов **setw**, **setiosflags** и **resetiosflags** и функций-элементов **setf** и **unsetf** для управления левым и правым выравниванием целочисленных данных.

```
// fig21_22.cpp
// Применение правого и левого выравнивания.

#include <iostream.h>
#include <iomanip.h>

main ( )
{
   int x = 12345;

   cout << "Default is right justified:\n"
        << setw(10) << x
        << "\n\nUSING MEMBER FUNCTIONS\n"
        << "Use setf to set ios::left:\n" << setw(10);

   cout.setf(ios::left, ios::adjustfield);
   cout << x << "\nUse unsetf to restore default:\n";
   cout.unsetf(ios::left);
   cout << setw(10) << x
        << "\n\nUSING PARAMETERIZED STREAM MANIPULATORS"
        << "\nUse setiosflags to set ios::left:\n"
        << setw(10) << setiosflags(ios::left) << x
        << "\nUse resetiosflags to restore default:\n"
        << setw(10) << resetiosflags(ios::left)
        << x << endl;
   return 0;
}
```

```
Default is right justified:
     12345

USING MEMBER FUNCTIONS
Use setf to set ios::left:
12345
Use unsetf to restore default:
     12345

USING PARAMETERIZED STREAM MANIPULATORS
Use setiosflags to set ios::left:
12345
Use resetiosflags to restore default:
     12345
```

Рис. 21.22. Применение правого и левого выравнивания

Флаг **internal** указывает, что знак числа (или основание системы счисления, если флаг **ios::showbase** установлен; см. раздел 21.7.5) должен выравниваться по левому краю поля, значение числа должно быть выровнено по правому краю, а в оставшееся пустое место выводятся символы-заполнители. Флаги **left**, **right** и **internal** содержатся в статическом элементе данных **ios::adjustfield**. При установке флагов **left**, **right** и **internal** при помощи функции **setf** ей нужно в качестве второго аргумента передавать **ios::adjustfield**. Это

гарантирует, что только один из трех флагов выравнивания будет установлен (поскольку они взаимно исключают друг друга). На рис. 21.23 показан пример использования в программе потоковых манипуляторов **setiosflags** и **setw** для установки выравнивания по ширине. Обратите внимание на использование флага **ios::showpos**, устанавливающего принудительный вывод знака «плюс».

```
// fig21_23.cpp
// Вывод числа с выравниванием по ширине
// и принудительным знаком "плюс".

#include <iostream.h>
#include <iomanip.h>

main()
{
   cout << setiosflags(ios::internal | ios::showpos)
        << setw(10) << 123 << endl;

   return 0;
}
```

```
+   123
```

Рис. 21.23. Вывод числа с выравниванием по ширине и принудительным знаком «плюс»

21.7.4. Заполнение (fill, setfill)

Функция-элемент **fill** определяет символ-заполнитель, используемый при выравнивании данных в поле; если не определено иначе, для заполнения выводятся пробелы. Функция **fill** возвращает значение предшествующего заполняющего символа. Задать символ-заполнитель можно также при помощи манипулятора потока. Пример задания символа-заполнителя приведен на рис. 21.24, где для этой цели применяется функция-элемент **fill** и манипулятор **setfill**.

21.7.5. Флаги установки основания целых чисел (ios::dec, ios::oct, ios::hex, ios::showbase)

Статический элемент данных **ios::basefield** (аналогичный используемому с **setf ios::adjustfield**) включает в себя биты **oct**, **hex** и **dec**, определяющие, что целые числа должны обрабатываться соответственно как восьмеричные, шестнадцатеричные и десятичные значения. Значение по умолчанию для операции помещения в поток — десятичные числа, если ни один из этих битов не установлен; извлекаемые из потока данные по умолчанию обрабатываются соответственно форме, в которой они вводятся: целые числа, начинающиеся с **0**, обрабатываются как восьмеричные, целые числа, начинающиеся с **0x** или **0X**, трактуются как шестнадцатеричные значения, а все другие целые числа предполагаются десятичными. Но как только для потока определено какое-то специфическое значение основания, все целые числа в этом потоке считаются записанными в установленной системе счисления, пока не будет определено новое значение основания, или до конца программы.

```
// fig21_24.cpp
// Использование функции-элемента fill и манипулятора setfill
// для замены символа заполнителя
// полей, больших по ширине, чем выводимые значения.

#include <iostream.h>
#include <iomanip.h>

main()
{
   int x = 10000;

   cout << x
        << " printed as int right and left justified\n"
        << "and as hex with internal justification.\n"
        << "Using the default pad character (space):\n";
   cout.setf(ios::showbase);
   cout << setw(10) << x << '\n';
   cout.setf(ios::left, ios::adjustfield);
   cout << setw(10) << x << '\n';
   cout.setf(ios::internal, ios::adjustfield);
   cout << setw(10) << hex << x << "\n\n";

   cout << "Using various padding characters:\n";
   cout.setf(ios::right, ios::adjustfield);
   cout.fill('*');
   cout << setw(10) << dec << x << '\n';
   cout.setf(ios::left, ios::adjustfield);
   cout << setw(10) << setfill('%') << x << '\n';
   cout.setf(ios::internal, ios::adjustfield);
   cout << setw(10) << setfill('^') << hex
        << x << endl;

   return 0;
}
```

```
10000 printed as int right and left justified
and as hex with internal justification.
Using the default pad character (space):
     10000
10000
0x    2710

Using various padding characters:
*****10000
10000%%%%%
0x^^^^^2710
```

Рис. 21.24. Использование функции-элемента **fill** и манипулятора **setfill** для замены символа заполнннения полей, больших по ширине, чем выводимые значения

Установите флаг **showbase** для принудительного указания основания при выводе целого числа. Десятичные числа при этом будут выводится как обычно, восьмеричные числа выводятся с ведущим **0**, а при выводе шестнадцатеричных чисел перед числом выводятся символы **0x** или **0X** (регистр определяется значением флага **uppercase**). На рис. 21.25 показано использование флага **showbase** при выводе целого числа в десятичном, восьмеричном и шестнадцатеричном форматах.

```
// fig21_25.cpp
// Применение флага ios::showbase

#include <iostream.h>
#include <iomanip.h>

main()
{
   int x = 100;

   cout << setiosflags(ios::showbase)
        << "Printing integers preceded by their base:\n"
        << x << '\n'
        << oct << x << '\n'
        << hex << x << endl;

   return 0;
}
```

```
Printing integers preceded by their base:
100
0144
0x64
```

Рис. 21.25. Применение флага **ios::showbase**

21.7.6. Числа с плавающей точкой; научная нотация (ios::scientific, ios::fixed)

Флаги **ios::scientific** и **ios::fixed** содержатся в статическом элементе данных **ios::floatfield**, аналогичном статическим данным-элементам **ios::adjustfield** и **ios::basefield**, используемым функцией **setf**. Эти флаги управляют форматом вывода чисел с плавающей точкой. Если установлен флаг **scientific**, то числа с плавающей точкой будут выводиться в научном формате. Флаг **fixed** устанавливает для чисел с плавающей точкой формат с фиксированным числом цифр после запятой (число цифр определяется функцией-элементом **precision**). Если ни один из этих флагов не установлен, то выходной формат определяется значением выводимого числа.

В результате вызова **cout.setf(0, ios::floatfield)** восстанавливается системное значение по умолчанию для вывода чисел с плавающей точкой. На рис. 21.26 показан пример вывода чисел с плавающей точкой в фиксированном и научном форматах.

21.7.7. Управление верхним/нижним регистрами (ios::uppercase)

При установленном флаге **ios::uppercase** символы **X** и **E**, используемые в *шестнадцатеричном и научном форматах, выводятся в верхнем регистре* (рис. 21.27). Кроме того, в этом случае все буквы, используемые в шестнадцатеричном представлении числа, будут выводиться в верхнем регистре.

```
// fig21_26.cpp
// Вывод чисел с плавающей точкой в формате по умолчанию,
// научном и фиксированном форматах.
#include <iostream.h>

main()
{
   double x = .001234567, y = 1.946e9;

   cout << "Displayed in default format:\n"
        << x << '\t' << y << '\n';
   cout.setf(ios::scientific, ios::floatfield);
   cout << "Displayed in scientific format:\n"
        << x << '\t' << y << '\n';
   cout.unsetf(ios::scientific);
   cout << "Displayed in default format after unsetf:\n"
        << x << '\t' << y << '\n';
   cout.setf(ios::fixed, ios::floatfield);
   cout << "Displayed in fixed format:\n"
        << x << '\t' << y << endl;

   return 0;
}
```

```
Displayed in default format:
0.001235  1.946e+09
Displayed in scientific format:
1.23567e-03  1.946e+09
Displayed in default format after unsetf:
0.001235  1.946e+09
Displayed in fixed format:
0.001235  1946000000
```

Рис. 21.26. Вывод чисел с плавающей точкой по умолчанию, в научном и в фиксированном форматах

```
// fig21_27.cpp
// Использование флага ios::uppercase
#include <iostream.h>
#include <iomanip.h>

main()
{
   cout << setiosflags(ios::uppercase)
        << "Printing uppercase letters in scientific\n"
        << "notation exponents and hexadecimal values:\n"
        << 4.345e10 << '\n'
        << hex << 123456789 << endl;

   return 0;
}
```

```
Printing uppercase letters in scientific
notation exponents and hexadecimal values:
4.345E+10
75BCD15
```

Рис. 21.27. Использование флага **ios::uppercase**

21.7.8. Установка и сброс флагов форматирования (flags, setiosflags, resetiosflags)

Функция-элемент **flags** без аргумента просто возвращает (как длинное целое) текущее состояние флагов форматирования. Функция-элемент **flags**, вызванная с аргументом типа **long**, устанавливает флаги формата в соответствии с их значениями в аргументе, и возвращает предыдущее состояние флагов. Флаги формата, не установленные в аргументе функции **flags**, сбрасываются. Программа на рис. 21.28 демонстрирует применение функции-элемента **flags** для установки новых параметров форматирования с сохранением предыдущих параметров форматирования и последующим восстановлением первоначальных установок.

```
// fig21_28.cpp
// Демонстрация возможностей функции-элемента flags.
#include <iostream.h>

main()
{
   int i = 1000;
   double d = 0.0947628;

   cout << "The value of the flags variable is: "
        << cout.flags() << '\n'
        << "Print int and double in original format:\n"
        << i << '\t' << d << "\n\n";
   long originalFormat = cout.flags(ios::oct | ios::scientific);
   cout << "The value of the flags variable is: "
        << cout.flags() << '\n'
        << "Print int and double in a new format\n"
        << "specified using the flags member function:\n"
        << i << '\t' << d << "\n\n";
   cout.flags(originalFormat);
   cout << "The value of the flags variable is: "
        << cout.flags() << '\n'
        << "Print values in original format again:\n"
        << i << '\t' << d << endl;

return 0;
}
```

```
The value of the flags variable is: 1
Print int and double in original format:
1000    0.094763

The value of the flags variable is: 4040
Print int and double in a new format
specified using the flags member function:
1750    9.47628e-02

The value of the flags variable is: 1
Print values in original format again:
1000    0.094763
```

Рис. 21.28. Демонстрация возможностей функции-элемента **flags**

В следующем операторе функция-элемент **setf** устанавливает флаги форматирования, указываемые в аргументе, и возвращает предыдущее состояние всех флагов в формате длинного целого.

```
long previousFlagSettings =
   cout.setf(ios::showpoint | ios::showpos);
```

Если функция-элемент вызывается с двумя аргументами типа **long**, как в следующей строке,

```
cout.setf(ios::left, ios::adjustfield);
```

то сначала очищаются биты **ios::adjustfield** и затем устанавливается флаг **ios::left**. Эта версия функции **setf** используется с битовыми полями **ios::basefield** (содержит флаги **ios::dec**, **ios::oct** и **ios::hex**), **ios::floatfield** (содержит **ios::scientific** и **ios::fixed**) и **ios::adjustfield** (содержит **ios::left**, **ios::right** и **ios::internal**).

Функция-элемент **unsetf** сбрасывает указанные флаги и возвращает предыдущее состояние всех флагов.

21.8. Состояния ошибки потоков

Состояние потока может быть проверено при помощи битов, определенных в классе **ios** — базовом классе классов **istream**, **ostream** и **iostream**, которые мы используем для ввода/вывода.

Бит состояния **eofbit** автоматически устанавливается для потока ввода, когда в потоке встречается конец файла. Программа может использовать функцию-элемент **eof**, чтобы определить, был ли достигнут конец файла в потоке. Следующий вызов

```
cin.eof()
```

возвращает true, если в **cin** встретился конец файла и false в противном случае.

Бит **failbit** устанавливается, когда в потоке происходит ошибка форматирования, но символы при этом не теряются. Функция-элемент **fail** может быть использована для определения результата потоковой операции; обычно такую ошибку можно исправить.

Установленный бит **badbit** свидетельствует о том, что произошла ошибка, связанная с потерей данных. Функция-элемент **bad** может определить, произошла ли в потоке такого типа ошибка. Такая серьезная ошибка обычно неисправима.

Бит **goodbit** устанавливается, если для потока ни один из битов **eofbit**, **failbit** или **badbit** не установлен.

Функция-элемент **good** возвращает true, если функции **bad**, **fail** и **eof**, все возвращают false. Операции ввода/вывода должны выполняться только на «хороших» потоках.

Функция-элемент **rdstate** возвращает состояние потока. Например, вызов **cout.rdstate()** возвратит состояние потока **cout**, которое можно затем исследовать при помощи оператора **switch** и проверить флаги **ios::eofbit**, **ios::badbit**, **ios::failbit** и **ios::goodbit**. Но предпочтение в тестировании состояния потока следует отдать функциям-элементам **eof**, **bad**, **fail** и **good** — при использовании этих функций не приходится иметь дело непосредственно с обработкой битов.

Функция-элемент **clear** обычно используется для восстановления «рабочего» состояния потока, после чего на нем можно продолжать операции ввода/вывода. Заданный по умолчанию аргумент для **clear** — **ios::goodbit**, так что в результате вызова

```
cin.clear();
```

будут очищены биты потока **cin** и установлен **goodbit**. Оператор

```
cin.clear(ios::failbit)
```

вызовет установку **failbit**. Пользователю может понадобиться такая операция при возникновении ошибок ввода из **cin** определяемых пользователем типов. Имя **clear** (очистить) кажется в этом контексте неподходящим, но ошибки здесь нет.

Программа на рис. 21.29 иллюстрирует вызовы функций-элементов **rdstate**, **eof**, **fail**, **bad**, **good** и **clear**.

Функция-элемент **operator!** возвращает true, если установлен бит **badbit**, **failbit** или установлены оба этих бита. Функция-элемент **operator void *** возвращает false, если установлен бит **badbit**, **failbit** или установлены оба этих бита. Эти функции удобны при обработке файлов.

21.9. Ввод/вывод определяемых пользователем типов

C++ позволяет вводить и выводить стандартные типы данных, используя операцию извлечения из потока >> и операцию << передачи в поток. Эти перегруженные операции могут обрабатывать все стандартные типы данных, включая адреса памяти и строки. Но программист может перегрузить операции передачи в поток и извлечения из потока, чтобы иметь возможность выполнять ввод/вывод определяемых пользователем типов. На рис. 21.30 показана перегрузка операций извлечения и передачи для обработки данных определяемого пользователем класса телефонных номеров **PhoneNumber**. Обратите внимание, что эта программа предполагает, что номера телефонов вводятся правильно. Мы оставляем проверку ошибок ввода для упражнения.

Определенная в программе на рис. 21.30 перегруженная операция извлечения из потока используется для ввода телефонного номера в виде

```
(800) 555-1212
```

в объекты типа **PhoneNumber**. Операция извлечения считывает три части номера телефона и помещает их в элементы **areaCode**, **exchange** и **line** объекта класса **PhoneNumber** (с именем **num** в функции-операции и **phone** в функции **main**). Скобки, пробелы и символы дефиса отбрасываются при вызове функции-элемента **ignore** класса **istream**. Функция-операция возвращает ссылку на объект **input** класса **istream**. Поскольку операция возвращает ссылку на объект-поток, операции извлечения объектов **PhoneNumber** могут конкатенироваться с операциями извлечения других объектов **PhoneNumber** и объектов других типов. Например, два объекта **PhoneNumber** можно ввести следующим образом:

```
cin >> phone1 >> phone2;
```

Операция передачи в поток имеет два аргумента: ссылку на тип **ostream** и ссылку на определяемый пользователем тип (в данном случае — **PhoneNumber**), и возвращает ссылку на тип **ostream**. В программе на рис. 21.30 перегруженная операция помещения в поток выводит объекты типа **PhoneNumher** в той же самой форме, в какой они вводились. Операция выводит части номера телефона как строки символов, потому что они хранятся в формате строки (напомним, что функция-элемент **getline** класса **istream** дописывает в конец строки нуль-символ после того, как завершает ввод).

```
// fig21_29.cpp
// Проверка состояния потока.

#include <iostream.h>

main()
{
   int x;
   cout << "Before a bad input operation:\n"
        <<  "cin.rdstate(): " << cin.rdstate()
        << "\n  cin.eof(): " << cin.eof()
        << "\n   cin.fail(): " << cin.fail()
        << "\n  cin.bad(): " << cin.bad()
        << "\n cin.good(): " << cin.good()
        << "\n\nExpects an integer, but enter a character: ";
   cin >> x;

   cout << "\nAfter a bad input operation:\n"
        <<  "cin.rdstate(): " << cin.rdstate()
        << "\n  cin.eof(): " << cin.eof()
        << "\n  cin.fail(): " << cin.fail()
        << "\n  cin.bad(): " << cin.bad()
        <<  "\n cin.good(): " << cin.good() << "\n\n";

   cin.clear();

   cout << "After cin.clear()\n"
        << "cin.fail (): " << cin.fail() << endl;

   return 0;
}
```

```
Before a bad input operation:
cin.rdstate(): 0
    cin.eof(): 0
   cin.fail(): 0
    cin.bad(): 0
   cin.good(): 1

Expects an integer, but enter a character: A

After a bad input operation:
cin.rdstate(): 2
    cin.eof(): 0
   cin.fail(): 2
    cin.bad(): 0
   cin.good(): 0

After cin.clear()
cin.fail (): 0
```

Рис. 21.29. Проверка состояния потока

Перегруженные функции-операции объявлены в классе **PhoneNumber** как дружественные функции. Эти функции должны быть дружественными, чтобы иметь доступ к закрытым элементам класса.

```
// fig21_30.cpp
// Определяемые пользователем операции извлечения и помещения

#include <iostream.h>

class PhoneNumber {
   friend ostream& operator<<(ostream&, PhoneNumber&);
   friend istream& operator>>(istream&, PhoneNumber&);
private:
   char areaCode[4];
   char exchange[4];
   char line[5];
};

ostream& operator<< (ostream& output, PhoneNumber& num)
{
   output << "(" << num.areaCode << ") "
          << num.exchange << "-" << num.line;

   return output;
}

istream& operator>>(istream& input, PhoneNumber& num)
{
   input.ignore();                    // пропуск (
   input.getline(num.areaCode, 4);
   input.ignore(2);                   // пропуск ) и пробела
   input.getline(num.exchange, 4);
   input.ignore();                    // пропуск -
   input.getline(num.line, 5);

   return input;
}

main ()
{
   PhoneNumber phone;

   cout << "Enter a phone number in the "
        <<  "form (123) 456-7890:\n";
   cin >> phone;
   cout << "The phone number entered was:\n"
        <<  phone << endl;

   return 0;
}
```

```
Enter a phone number in the form (123) 456-7890:
(800) 555-1212
The phone number entered was:
(800) 555-1212
```

Рис. 21.30. Определяемые пользователем операции извлечения и передачи.

Общее методическое замечание 21.3

Для того, чтобы ввести в C++ возможности ввода/вывода новых, определяемых пользователем типов данных, не требуется вносить изменения в закрытые секции данных классов **ostream** и **istream**. Расширяемость языка C++ — одна из его наиболее привлекательных черт.

21.10. Привязка потока вывода к потоку ввода

Интерактивные прикладные программы обычно используют класс **istream** для ввода данных и **ostream** — для вывода. Часто пользователь вводит необходимые данные в ответ на приглашение программы. Очевидно, что текст приглашения должен появляться перед операцией ввода данных, но так как операции вывода буферизованы, то вывод появляется только тогда, когда буфер заполнен, или когда буфер очищается явно программой, или сбрасывается автоматически по окончании ее работы. В C++ имеется функция-элемент **tie** для синхронизации, т.е. «привязки», действий классов **istream** и **ostream**, вызов которой гарантирует, что вывод появится перед связанным с ним вводом. В следующем примере вызова:

```
cin.tie(cout);
```

поток **cout (ostream)** привязывается к потоку **cin (istream)**. Приведенный пример вызова функции привязки потоков избыточен, потому что C++ выполняет эту операцию автоматически, обеспечивая стандартный интерфейс пользователя. Однако пользователю может потребоваться связывание других пар потоков **istream** и **ostream**. Чтобы «отвязать» входной поток **inputStream** от выходного потока, используется вызов

```
inputStream.tie(0);
```

Резюме

- Операции ввода/вывода выполняются в соответствии с типом данных.
- В C++ ввод/вывод данных производится потоками байтов. Поток — это просто байтовая последовательность.
- Работа системных механизмов ввода/вывода сводится к перемещению байтов данных от устройств в память и обратно, обеспечивая при этом эффективность и надежность.
- В C++ имеются возможности для ввода/вывода «высокого уровня» и «низкого уровня». Низкоуровневый ввод/вывод обычно занимается передачей некоторого числа байт от устройства в память или из памяти — устройству. Функции ввода/вывода высокого уровня оперируют байтами, сгруппированными в элементы данных вроде целых чисел, чисел с плавающей точкой, символов, строк и определяемых пользователем типов.
- C++ имеет средства неформатируемого и форматируемого ввода/вывода. Неформатируемый ввод/вывод выполняется быстро, но обрабатывает сырые данные, которые неудобны для восприятия их человеком. Форматируемый ввод/вывод обрабатывает данные в виде имеющих смысл групп байтов, но требует больше времени для своего выполнения, что может негативно сказаться при обработке данных большого объема.

- Большинство программ C++ включает файл заголовка **iostream.h**, который содержит основную информацию, необходимую для всех операций потокового ввода/вывода.
- В файле-заголовке **iomanip.h** содержится информация, необходимая для форматируемого ввода/вывода с параметризованными манипуляторами потоков.
- Заголовок **fstream.h** содержит данные, нужные для операций обработки файлов.
- Заголовок **strstream.h** содержит информацию, необходимую для операций форматирования в памяти.
- Файл заголовка **stdiostream.h** необходим для программ, в которых совмещаются ввод/вывод в стиле C и потоки C++.
- Класс **istream** поддерживает операции ввода из потока.
- Класс **ostream** поддерживает операции вывода в поток.
- Класс **iostream** поддерживает как операции потокового ввода, так и операции вывода.
- Оба класса **istream** и класс **ostream** выведены путем простого наследования из базового класса **ios**.
- Класс **iostream** выведен путем сложного наследования из классов **istream** и **ostream**.
- Перегруженная операция левого сдвига (<<) обозначает вывод в поток и называется операцией передачи в поток.
- Перегруженная операция правого сдвига (>>) обозначает ввод из потока и называется операцией извлечения из потока.
- Объект **cin** класса **istream** ассоциирован со стандартным устройством ввода данных, обычно с клавиатурой.
- Объект **cout** класса **ostream** ассоциируется со стандартным устройством вывода, обычно это экран монитора.
- Объект **cerr** класса **ostream** ассоциирован со стандартным устройством ошибки. Вывод в **cerr** не буферизован; результат каждой операции передачи в **cerr** появляется немедленно.
- Манипулятор потока **endl** передает в поток символ новой строки и сбрасывает буфер вывода.
- Компилятор C++ определяет типы данных при вводе/выводе автоматически.
- По умолчанию адреса отображаются в шестнадцатеричном формате.
- Чтобы вывести адрес, находящийся в переменной-указателе типа **char ***, приведите указатель к типу **void ***.
- Функция-элемент **put** выводит один символ. Возможна конкатенация вызовов функции **put**.
- Ввод из потока выполняется при помощи операции >> извлечени из потока. Эта операция автоматически пропускает во входном потоке пробельные символы.

- Операция >> возвращает false, когда в потоке встречается конец файла.
- При операциях извлечения из потока **failbit** устанавливается, когда возникает ошибка во входных данных, а **badbit** устанавливается, если возникает серьезная ошибка.
- Ряд значений можно вводить из потока, поместив операцию извлечения в заголовке цикла **while**. Операция извлечения возвращает false (нуль), когда в потоке встречается конец файла.
- Функция-элемент **get** без аргументов вводит один символ и возвращает его в качестве своего значения; при вводе конца файла функция **get** возвращает **EOF**.
- Функция-элемент **get** с символьным аргументом извлекает символ из потока ввода. Эта версия функции **get** возвращает значение false при появлении конца файла; в противном случае эта версия возвращает ссылку на объект класса **istream**, для которого она вызывалась.
- Версия функции-элемента **get** с тремя аргументами получает символьный массив, размер массива и символ-ограничитель (по умолчанию это символ новой строки). Эта версия считывает символы из входного потока следующим образом: вводится на один символ меньше, чем указанный во втором аргументе размер массива, после чего ввод прекращается, либо ввод прекращается раньше, если встретится символ-ограничитель. В конец строки, вводимой в символьный массив, дописывается нуль-символ. Ограничитель не записывается в символьный массив, а остается в потоке ввода.
- Функция-элемент **getline** работает подобно третьей версии функции-элемента **get**, но удаляет символ-разделитель из потока.
- Функция-элемент **ignore** пропускает определенное количество символов (по умолчанию — один символ) во входном потоке; выполнение завершается досрочно, если встречается символ-ограничитель (по умолчанию — **EOF**).
- Функция-элемент **putback** возвращает обратно в поток предыдущий символ, полученный из потока функцией **get**.
- Функция-элемент **peek** извлекает следующий символ из входного потока, но не удаляет его из потока.
- В C ++ реализован безопасный по типу ввод/вывод. При обработке операциями << и >> недопустимого типа данных устанавливаются различные флаги ошибок, которые пользователь может проверить, чтобы определить, завершилась ли операция ввода/вывода успешно или неудачно.
- Неформатируемый ввод/вывод выполняется функциями-элементами **read** и **write**. Каждая из этих функций вводит или выводит некоторое число байт в память или из памяти, используя для этого указанный адрес. Эти функции вводят/выводят сырые данные, которые никак не форматируются.
- Функция-элемент **gcount** возвращает число символов, введенных из потока предыдущим вызовом **read**.

- Функция-элемент **read** вводит заданное число символов в символьный массив. Если было введено меньшее количество символов, устанавливается бит **failbit**.
- Для того, чтобы изменить основание системы счисления, в которой целые числа будут помещаться в поток, применяется манипулятор **hex** — для шестнадцатеричной системы (по основанию 16) и манипулятор **oct** — для восьмеричной (по основанию 8). Манипулятор потока **dec** возвращает поток к десятичной системе счисления. Установленное в потоке основание остается в силе, пока не будет явно изменено.
- Основание системы счисления в потоке также может быть изменено манипулятором потока **setbase**, который имеет один целочисленный аргумент, принимающий значения **10**, **8** или **16** для соответствующих систем счисления.
- Можно управлять точностью представления чисел с плавающей точкой, используя манипулятор потока **setprecision** или функцию-элемент **precision**. Любым из этих способов точность устанавливается для всех последующих операций вывода до следующей установки. Функция-элемент **precision** без аргумента возвращает текущую установку точности. Точность **0** устанавливает точность, заданную по умолчанию (равную **6**).
- При использовании параметризованных манипуляторов необходимо включить в программу файл заголовка **iomanip.h**.
- Функция-элемент **width** устанавливает ширину поля и возвращает предыдущее значение этой величины. В случае, если обрабатываемые значения требуют места меньше, чем ширина поля, в пустые позиции выводится символ-заполнитель. Установленное значение ширины применяется только для последующей операции помещения или извлечения, после чего ширина поля устанавливается равной **0**, т.е. выводимые значения будут занимать поля ширины, необходимой для вывода текущего значения. Если величина требует для своего размещения места больше, чем отведенная ширина поля, усечение не производится, значение выводится полностью. Функция **width** без аргумента возвращает текущее значение ширины поля. Для установки ширины поля можно также использовать манипулятор потока **setw**.
- При вводе манипулятор потока **setw** устанавливает максимальный размер вводимой строки; если вводится строка большего размера, то она разбивается на части, не превышающие размер, установленный **setw**.
- Пользователи могут создавать свои собственные манипуляторы потока.
- Функции-элементы **setf**, **unsetf** и **flags** управляют значениями флагов.
- Флаг **skipws** указывает, что операция >> должна пропускать во входном потоке пробельные символы. Манипулятор потока **ws** также заставляет игнорировать во входном потоке ведущие пробельные символы.
- Флаги форматирования определены в классе **ios**.
- Флагами форматирования можно управлять с помощью функций-элементов **flags** и **setf**, но многие программисты предпочитают пользоваться манипуляторами потока. Программист может применить опера-

цию побитового ИЛИ |, чтобы объединить различные опции в одном числе типа **long**. При вызове функции-элемента **flags** с таким набором битов устанавливаются новые опции форматирования для данного потока и возвращается значение типа **long**, содержащее предыдущее состояние битов. Это значение обычно сохраняется, для того чтобы потом можно было вызвать **flags** с этим значением и восстановить предыдущие опции потока.

- Функция **flags** должна вызываться со значением, представляющим состояние всех флагов. Функция **setf** с одним аргументом, с другой стороны, объединяет по ИЛИ один или большее количество флагов с существующими значениями флагов, формируя таким образом новое состояние форматирования потока.
- При установленном флаге **showpoint** вещественные числа выводятся с десятичной точкой и заданным числом цифр после десятичной точки, определяемым точностью представления.
- Флаги **left** и **right** позволяют выводить данные с левым выравниванием и заполнением правой части поля вывода символами-заполнителями, или с правым выравниванием и заполняющими символами слева соответственно.
- Флаг **internal** указывает, что знак числа (или основание системы счисления, если флаг **ios::showbase** установлен) должен выравниваться по левому краю поля, значение числа должно быть выровнено по правому краю, а в оставшиеся пустые позиции выводится символ-заполнитель.
- Элемент данных **ios::adjustfield** содержит флаги **left**, **right** и **internal**.
- Функция-элемент **fill** определяет символ-заполнитель, используемый при **left**-, **right**- и **internal**-выравнивании данных в поле (по умолчанию это пробел). Функция **fill** возвращает значение предшествующего заполняющего символа. Задать символ-заполнитель можно также при помощи манипулятора потока **setfill**.
- Статический элемент данных **ios::basefield** включает в себя биты **oct**, **hex** и **dec**, определяющие, что целые числа должны обрабатываться соответственно как восьмеричные, шестнадцатеричные и десятичные значения. По умолчанию операция передачи в поток выводит числа в десятичном виде, если ни один из этих битов не установлен; извлекаемые из потока данные по умолчанию обрабатываются соответственно форме, в которой они вводятся.
- Установите бит **showbase**, чтобы при выводе целых чисел указывалось основание системы счисления.
- Флаги **scientific** и **fixed** содержатся в статическом элементе данных **ios::floatfield**. Если установлен флаг **scientific**, то числа с плавающей точкой будут выводиться в научном формате. Флаг **fixed** устанавливает для чисел с плавающей точкой формат с фиксированным числом цифр после запятой; число цифр определяется функцией-элементом **precision**.
- Вызов функции **cout.setf(0, ios::floatfield)** восстанавливает заданный по умолчанию формат для чисел с плавающей точкой.

- При установленном флаге **uppercase** символы **X** и **E**, используемые в шестнадцатеричном и научном форматах, выводятся в верхнем регистре. Кроме того, все буквы, используемые в шестнадцатеричном представлении числа, также будут выводиться в верхнем регистре.
- Функция-элемент **flags** без аргумента просто возвращает (как длинное целое) текущее состояние флагов форматирования. Функция-элемент **flags**, вызванная с аргументом типа **long**, устанавливает флаги формата, заданные аргументом, и возвращает предыдущее состояние флагов.
- Функция-элемент **setf** устанавливает флаги форматирования, указываемые в аргументе, и возвращает предыдущее состояние флагов в формате длинного целого.
- Функция-элемент **setf**, вызываемая с двумя аргументами типа **long** — битом флага и битовым полем, — очищает биты в битовом поле и затем устанавливает флаг, передаваемый в первом аргументе.
- Функция-элемент **unsetf** сбрасывает указанные флаги и возвращает предыдущее состояние флагов.
- Параметризованный манипулятор потока **setiosflags** выполняет те же самые действия, что и функция-элемент **flags**.
- Параметризованный манипулятор потока **resetiosflags** выполняет те же самые действия, что и функция-элемент **unsetf**.
- Состояние потока можно узнать, просмотрев биты состояния, определенные в классе **ios**.
- Бит состояния **eofbit** устанавливается для потока ввода, когда операция ввода встречает конец файла. Для проверки состояния бита **eofbit** можно использовать функцию-элемент **eof**.
- Бит **failbit** устанавливается, когда в потоке происходит ошибка форматирования; символы при этом не теряются. Функция-элемент **fail** также может быть использована для определения, не произошла ли такого рода ошибка; обычно такую ошибку можно исправить.
- Установленный бит **badbit** свидетельствует о том, что произошла ошибка, связанная с потерей данных. Функция-элемент **bad** также может определить, произошла ли такого рода ошибка в потоке. Такая серьезная ошибка обычно неисправима.
- Функция-элемент **good** возвращает true, если функции **bad**, **fail** и **eof** все возвращают false. Операции ввода/вывода должны выполняться только на «хороших» потоках.
- Функция-элемент **rdstate** возвращает состояние потока.
- Функция-элемент **clear** обычно используется для восстановления «хорошего» состояния потока, чтобы на нем можно было продолжать операции ввода/вывода.
- Пользователь может перегрузить операции передачи в поток и извлечения из потока, чтобы иметь возможность выполнять ввод/вывод определяемых пользователем типов.

- Перегруженная операция извлечения из потока принимает в качестве аргументов ссылку на объект **istream** и ссылку на определяемый пользователем тип данных, и возвращает ссылку на объект **istream**.
- Перегруженная операция передачи в поток принимает ссылку на объект **ostream** и ссылку на определяемый пользователем тип данных и возвращает ссылку на объект **ostream**.
- Перегруженные функции потоковых операций часто объявляются как дружественные, чтобы иметь доступ к не-открытым элементам класса.
- В C++ имеется функция-элемент **tie** для синхронизации действий с классами **istream** и **ostream**, вызов которой гарантирует, что вывод появится перед связанным с ним вводом.

Терминология

badbit
cerr
cin
clog
cout
endl
eofbit
failbit
ios::adjustfield
ios::basefield
ios::fixed
ios::floatfield
ios::internal
ios::scientific
ios::showbase
ios::showpoint
ios::showpos
left
skipws
uppercase
width
безопасный по типу ввод/вывод
библиотека классов потоков
восьмеричные числа
(начинаются с **0**)
заполнение
класс **fstream**
класс **ifstream**
класс **ios**
класс **iostream**
класс **istream**
класс **ofstream**
класс **ostream**
конец файла
левое выравнивание
манипулятор потока
манипулятор потока **dec**
манипулятор потока **flush**
манипулятор потока **hex**
манипулятор потока **oct**
манипулятор потока **resetiosflags**
манипулятор потока **setbase**
манипулятор потока **setfill**
манипулятор потока **setprecision**
манипулятор потока **setw**
неформатированный ввод/вывод
операция извлечения из потока (>>)
операция передачи в поток (<<)
параметризованный манипулятор потока
потоки, определяемые пользователем
потоковый ввод
потоковый вывод
правое выравнивание
предопределенные потоки
пробельные символы
расширяемость
символ-заполнитель
символ-заполнитель по умолчанию
(пробел)
состояния формата
стандартный заголовочный файл
<iomanip.h>
точность по умолчанию
флаги формата
форматирование в памяти
форматированный ввод/вывод
функция-элемент **bad**
функция-элемент **clear**
функция-элемент **eof**
функция-элемент **fail**
функция-элемент **fill**
функция-элемент **flags**
функция-элемент **flush**
функция-элемент **gcount**
функция-элемент **get**
функция-элемент **getline**
функция-элемент **good**
функция-элемент **ignore**
функция-элемент ***operator void****
функция-элемент **operator!**

функция-элемент **peek**
функция-элемент **precision**
функция-элемент **put**
функция-элемент **putback**
функция-элемент **rdstate**
функция-элемент **read**
функция-элемент **setf**
функция-элемент **tie**
функция-элемент **unsetf**
функция-элемент **write**
функция-элемент **ws**
шестнадцатеричные числа (начинаются с **0x** или **0X**)
ширина поля

Распространенные ошибки программирования

21.1. Попытка чтения данных из потока **ostream** (или любого другого потока вывода).

21.2. Попытка записи данных в поток **istream** (или любой другой поток ввода).

21.3. В операторе ввода/вывода из потока вы опускают скобки, и в результате из-за относительно высокого приоритета операций помещения << и извлечения >> операции выполняются не в нужном порядке.

21.4. Если установленное значение ширины поля меньше, чем необходимо для вывода, выводимая величина будет помещена в поле необходимой ширины, но при этом результат вывода будет неудобочитаем или будет восприниматься неверно.

Хороший стиль программирования

21.1. В программах на C++ используйте ввод/вывод только в стиле языка C++ и не используйте функции ввода/вывода из библиотеки C, хотя они и доступны в C++.

21.2. При выводе выражений заключайте их в скобки, чтобы избежать конфликтов, связанных со старшинством операции << и операций в выражении.

Советы по повышению эффективности

21.1. Применяйте неформатируемый ввод/вывод при обработке файлов большого объема.

Общие методические замечания

21.1. Ввод/вывод в C++ безопасен в отношении использования различных типов.

21.2. C ++ предлагает единообразный интерфейс ввода/вывода для предопределенных типов и типов, определяемых пользователем. Благодаря такому общему подходу облегчается процесс разработки программ и, в частности, повторное использование программного обеспечения.

21.3. Для того, чтобы ввести в C++ возможности ввода/вывода новых, определяемых пользователем типов данных, не требуется вносить изменения в закрытые секции данных классов ostream и istream. Расширяемость языка C++ — одна из его наиболее привлекательных черт.

Упражнения для самоконтроля

21.1. Заполнив пропуски в следующих утверждениях :

a) Перегруженные функции потоковых операций должны быть объявлены как _____ функции класса.

b) Биты форматирования, которые отвечают за выравнивание, включают в себя _____, _____ и _____.

c) Ввод/вывод в C++ рассматривается как _____ байтов.

d) Параметризованные манипуляторы потока _____ и _____ могут быть использованы для установки и сброса флагов форматирования.

e) Большинство программ C ++ должно включать файл заголовка _____, который содержит основную информацию, необходимую для всех потоковых операций ввода/вывода.

f) Функции-элементы _____ и _____ могут использоваться для установки и сброса флагов форматирования.

g) Файл заголовка _____ содержит информацию, необходимую для выполнения форматирования в памяти.

h) При использовании параметризованных манипуляторов в программу должен быть включен файл заголовка _____.

i) В файле заголовка _____ содержится информация, необходимая для выполнения управляемой пользователем обработки файлов.

j) Манипулятор потока _____ помещает символ новой строки в выходной поток и сбрасывает буфер потока вывода.

k) Файл заголовка _____ содержит информацию, нужную программам, которые совмещают ввод/вывод C и C++.

l) Функция-элемент _____ класса **ostream** используется для выполнения неформатируемого вывода.

m) Операции ввода поддерживаются классом _____.

n) Вывод на стандартный поток ошибок направляется или на объект-поток _____, или на объект _____.

o) Операции вывода поддерживаются классом _____.

p) Для операции передачи в поток используется знак _____.

q) Четыре объекта, соответствующих стандартным системным устройствам, — это _____, _____, _____ и _____.

r) Для операции извлечения из потока используется знак _____.

s) Для вывода целых чисел в восьмеричном, шестнадцатеричном и десятичном форматах используются потоковые манипуляторы _____, _____ и _____.

t) Заданная по умолчанию точность представления чисел с плавающей точкой равна _____.

u) При установленном флаге _____ положительные числа выводятся со знаком «плюс».

21.2. *Являются ли следующие утверждения верными или нет? Если утверждение неверно, объясните, почему.*

a) Потоковая функция-элемент **flags()** с аргументом типа **long** устанавливает соответствующее своему аргументу значение переменной состояния потока **flags** и возвращает ее предыдущее значение.

b) Перегруженные операции передачи (<<) и извлечения из потока (>>) обрабатывают все предопределенные типы данных, включая строки, адреса памяти и определяемые пользователем типы данных.

c) Потоковая функция-элемент **flags()** без аргументов сбрасывает все флаговые биты в переменной состояния потока **flags**.

d) При перегрузке операции >> извлечения из потока функция-операция получает ссылку на тип **istream** и ссылку на определяемый пользователем тип и возвращает ссылку на **istream**.

e) Манипулятор потока **ws** пропускает ведущие пробельные символы в потоке ввода.

f) При перегрузке операции << помещения в поток функция-операция принимает ссылку на тип **istream** и ссылку на определяемый пользователем тип и возвращает ссылку на **istream**.

g) Операция >> извлечения из потока всегда пропускает во входном потоке ведущие пробельные символы.

h) Обработка ввода/вывода является частью языка C++.

i) Функция-элемент потока **rdstate()** возвращает текущее состояние потока.

j) Поток **cout** по умолчанию ассоциируется с экраном монитора.

k) Потоковая функция-элемент **good()** возвращает true, если функции-элементы **bad()**, **fail()** и **eof()** все возвращают false.

l) Поток **cin** по умолчанию ассоциируется с экраном монитора.

m) Если во время операции потокового ввода/вывода происходит неустранимая ошибка, функция-элемент **bad()** возвращает true.

n) Вывод на **cerr** не буферизован, а вывод **clog** — буферизован.

o) Когда флаг **ios::showpoint** установлен, то значения с плавающей точкой выводятся по умолчанию с шестью цифрами после запятой, при условии, что значение точности не изменялось, в противном случае числа выводятся с установленной точностью.

p) Функция-элемент **put** класса **ostream** выводит заданное количество символов.

q) Действие потоковых манипуляторов **dec**, **oct** и **hex** распространяется только на следующую операцию вывода целого числа.

r) Адреса памяти по умолчанию выводятся в формате длинного целого.

21.3. *Для выполнения каждого из следующих действий напишите по одному оператору языка C++.*

a) Выведите строку "**Enter your name:** ".

b) Установите флаг, отвечающий за вывод экспоненты в научной нотации и букв в шестнадцатеричных числах в верхнем регистре.

c) Выведите адрес, записанный в переменной **string** типа **char ***.

d) Установите нужный флаг, чтобы числа с плавающей точкой выводились научном формате.

e) Выведите адрес, записанный в переменной **integerPtr** типа **int ***.

f) Установите флаг, задающий режим добавления символов основания при выводе восьмеричных и шестнадцатеричных целочисленных значений.

g) Выведите значение, на которое ссылается указатель **floatPtr** типа **float ***.

h) Используя потоковую функцию-элемент, установите символ-заполнитель '*****' поля большей ширины, чем выводимое значение. Напишите оператор, выполняющий ту же задачу при помощи манипулятора потока.

i) Выведите символы '**O**' и '**K**' в одном операторе с помощью функции-элемента **put** класса **ostream**.

j) Получите символ из потока ввода, не удаляя его из этого потока.

k) Введите символ в переменную **c** типа **char** двумя различными способами, используя функцию-элемент **get** класса **istream**.

l) Введите и отбросьте шесть символов из потока ввода.

m) Введите 50 символов в массив **line** типа **char**, используя функцию-элемент **read** класса **istream**.

n) Считайте 10 символов в массив символов name. Ввод прекращается, если встречается ограничитель '.'. Не удаляйте его из потока ввода. Напишите другой оператор, который выполняет эту задачу и удаляет ограничитель из потока.

o) Используя функцию-элемент **gcount** класса **istream**, определите число введенных в символьный массив **line** символов при последнем обращении к функции-элементу **read** класса **istream** и выведите это число, вызвав функцию-элемент **write** класса **ostream**.

p) Напишите два оператора сброса (принудительного вывода данных) потока вывода, используя функцию-элемент и манипулятор потока.

q) Выведите следующие значения: **124**, **18.376**, '**Z**', **1000000** и "**String**".

r) Выведите текущую установку точности, используя функцию-элемент.

s) Введите целое значение в переменную **months** типа **int** и вещественное число в переменную **percentageRate** типа **float**.

t) Выведите **1.92**, **1.925** и **1.9258** с точностью, равной трем цифрам, установив ее с помощью манипулятора потока.

u) Используя манипуляторы потока, выведите целое число **100** в восьмеричном, шестнадцатеричном и десятичном форматах.

v) Выведите целое число **100** в десятичном, восьмеричном и шестнадцатеричном представлении, применив для этого один и тот же манипулятор установки основания.

w) Выведите число **1234** с выравниванием по правому краю в поле из **10** цифр.

x) Введите символы в символьный массив **line**, пока не встретится символ '**z**', но не более **20** символов (включая завершающий нуль-символ). Не извлекайте символ-разделитель из потока.

y) При помощи целых переменных **x** и **y** для задайте ширину поля и точность; используйте эти установки при выводе значения двойной точности **87.4573**.

21.4. Найдите ошибку в каждом из следующих операторов и объясните, как ее можно исправить.

a)
```
cout << "Value of x <= y is: " << x <= y;
```

b) Следующий оператор должен вывести числовое значение '**c**'.
```
cout << 'c';
```

c)
```
cout << "" A string in quotes "";
```

21.5. Для каждого из следующих фрагментов кода покажите, что будет выведено на экран.

a)
```
cout << "12345\n";
cout.width(5);
cout.fill('*');
cout << 123 << '\n' << 123;
```

b)
```
cout << setw(10) << setfill('$') << 10000;
```

c)
```
cout << setw(8) << setprecision(3) << 1024.987654;
```

d)
```
cout << setiosflags(ios::showbase) << oct << 99
     << '\n' << hex << 99;
```

e)
```
cout << 100000 << '\n'
     << setiosflags(ios::showpos) << 100000;
```

f)
```
cout << setw(10) << setprecision(2) <<
     << setiosflags(ios::scientific) << 444.93738;
```

Ответы на упражнения для самоконтроля

21.1. a) дружественные. b) **ios::left**, **ios::right** и **ios::internal**. c) потоки. d) **setiosflags**, **resetiosflags**. e) **iostream.h**. f) **setf**, **unsetf**. g) **strstream.h**. h) **iomanip.h**. i) **fstream.h**. j) **endl**. k) **stdiostream.h**. l) **write**. m) **istream**. n) **cerr** или **clog**. o) **ostream**. p) **<<**. q) **cin**, **cout**, **cerr** и **clog**. r) **>>**. s) **oct**, **hex**, **dec**. t) шесть цифр после запятой. u) **ios::showpos**.

21.2. a) Верно.

b) Неверно. Операции передачи и извлечения не перегружены для всех определяемых пользователем типов. Для каждого определяемого пользователем типа программист должен написать соответствующие перегруженные функции-операции.

c) Неверно. Функция-элемент **flags()** без аргументов просто возвращает текущее значение переменной состояния потока **flags**.

d) Верно.

e) Верно.

f) Неверно. При перегрузке операции передачи в поток << функция-операция должна принимать ссылку на тип **ostream** и ссылку на определяемый пользователем тип и возвращать ссылку на тип **ostream**.

g) Верно. Если флаг **ios::skipws** не сброшен.

h) Неверно. Возможности ввода/вывода в C++ обеспечиваются стандартными библиотеками C++. Сам язык не поддерживает операций ввода/вывода или обработки файлов.

i) Верно.

j) Верно.

k) Верно.

l) Неверно. Поток **cin** ассоциируется со стандартным устройством ввода компьютера, обычно — клавиатурой.

m) Верно.

n) Верно.

o) Верно.

p) Неверно. Функция-элемент **put** класса **ostream** выводит один символ.

q) Неверно. Потоковые манипуляторы **dec**, **oct** и **hex** устанавливают основание системы счисления целых чисел в потоке для всех последующих операций, пока другой манипулятор не изменит основание или программа не завершит свою работу.

r) Неверно. Адреса памяти по умолчанию выводятся в шестнадцатеричном формате. Чтобы вывести адрес в формате длинного целого, адресное выражение нужно преобразовать к типу длинного целого.

21.3. a) `cout << "Enter your name: ";`

b) `cout.setf(ios::uppercase);`

c) `cout << long(string);`

d) `cout.setf(ios::scientific, ios::floatfield);`

e) `cout << integerPtr;`

f) `cout << setiosflags(ios::showbase);`

g) `cout << *floatPtr;`

h)
```
cout.fill('*');
coutt << setfill('*');
```

i) `cout.put('0').put('K');`

j) `cin.peek();`

k)
```
c = cin.get();
cin.get(c);
```

l) `cin.ignore(6);`

m) `cin.read(line, 50);`

n)
```
cin.get(name, 10, '.');
cin.getline(name, 10, '.');
```

o) `cout.write(line, cin.gcount());`

p)
```
cout.flush();
cout << flush;
```

q) `cout << 124 << 18.376 << 'Z' << 1000000 << "String";`

r) `cout.precision();`

s) `cin >> months >> percentageRate;`

t)
```
cout << setprecision(3) << 1.92 << '\t'
     << 1.925 << '\t' << 1.9258;
```

u) `cout << oct << 100 << hex << 100 << dec << 100;`

v) `cout << 100 << setbase(8) << 100 << setbase(16) << 100;`

w) `cout << setw(10) << 1234;`

x) `cin.get(line, 20, 'z');`

y) `cout << setw(x) << setprecision(y) << 87.4573;`

21.4. a) Ошибка: приоритет операции << выше, чем приоритет <=, в результате чего оператор будет оцениваться неправильно и компилятор выдаст сообщение об ошибке.

Исправление: чтобы исправить оператор, заключите в круглые скобки выражение **x <= y**. Такая ошибка произойдет с любым выражением, в котором используются операции с меньшим приоритетом, чем у операции <<, если выражение не заключить в круглые скобки.

b) Ошибка: в C ++ символы не рассматриваются как короткие целые числа, как это было в языке C.

Исправление: чтобы вывести численное значение символа из таблицы допустимых символов компьютера, его нужно привести к целому типу, как в следующем выражении:

```
cout << int('c');
```

c) Ошибка: символы двойных кавычек не могут быть выведены в составе строки, если не использовать escape-код.

Исправление: выведите строку одним из следующих способов:

```
cout << '"' << "A string in quotes" << '"';
        cout << "\"A string in quotes\"";
```

21.5. a)
```
12345
**123
123
```

b)
```
$$$$$10000
```

c)
```
1024.988
```

d)
```
0143
0x63
```

e)
```
100000
+100000
```

f)
```
4.45e+02
```

Упражнения

21.6. Выполните каждое из следующих действий при помощи одного оператора языка:

a) Выведите целое число **40000** в поле из **15** цифр с левым выравниванием.

b) Введите строку в символьный массив **state**.

c) Выведите число **200** со знаком и без знака.

d) Выведите **100** в шестнадцатеричном формате с указанием основания **0x**.

e) Считайте символы в массив **s**, пока не встретится символ-ограничитель '**p**', но не более 10 символов (включая нуль-символ). Ограничитель должен быть извлечен из потока и отброшен.

f) Выведите число **1.234** в поле из **9** цифр с ведущими нулями.

g) Введите со стандартного потока ввода строку в форме "**characters**". Сохраните строку в символьном массиве **s**. Отбросьте символы кавычек из входного потока. Введите не более 50 символов (включая завершающий нуль-символ).

21.7. Напишите программу ввода целых чисел в десятичном, восьмеричном и шестнадцатеричном форматах. Выведите каждое прочитанное целое число во всех трех форматах. Проверьте программу со следующими входными данными: 10, 010, 0x10.

21.8. Напишите программу, которая выводит значения указателей, приведенных к различным целочисленным типам. В каких случаях вы получите странные значения? В каких случаях произойдут ошибки?

21.9. Напишите тестовую программу вывода целочисленного значения **12345** и вещественного числа **1.2345** в поля различной ширины. Что получается, когда значение выводится в поле, содержащее меньшее количество позиций, чем выводимое значение?

21.10. Напишите программу, которая выводит варианты округления числа **100.453627** до целого значения и до числа с дробной частью с десятыми, сотыми, тысячными и десятитысячными.

21.11. Напишите программу, которая вводит строку символов с клавиатуры и определяет ее длину. Выведите эту строку, задав в качестве ширины поля вывода удвоенную длину.

21.12. Напишите программу перевода температуры по шкале Фаренгейта — целое число, изменяющееся от **0** до **212** — в температуру по шкале Цельсия — вещественное число с точностью 3 цифры. Используйте формулу преобразования

```
celsius = 5.0/9.0 * (fahrenheit - 32);
```

Данные выведите в два столбца с правым выравниванием, причем значения температуры по Цельсию выводите со знаком как для отрицательных, так и для положительных значений.

21.13. В некоторых языках программирования строки могут заключаться в одиночные или двойные кавычки. Напишите программу, которая

вводит три строки **suzy**, **"suzy"** и **'suzy'**. Будут ли одиночные и двойные кавычки введены как часть строки или будут отброшены?

21.14. В программе на рис. 21.30 операции извлечения из потока и передачи в поток были перегружены, чтобы иметь возможность ввода/вывода объектов класса **PhoneNumber**. Перепишите функцию-операцию извлечения из потока, чтобы она выполняла описанную ниже процедуру проверки корректности ввода. Функцию-операцию >> для этого придется полностью переписать.

a) Введите сразу весь номер телефона в массив. Затем убедитесь в том, что введено было соответствующее число символов (символов должно быть 14, при записи номера телефона в форме **(800) 555-1212**). Используйте функцию-элемент потока **clear**, чтобы установить бит **ios::fail**, если ввод неверен.

b) Поскольку код региона и код телефонной станции в номере телефона не должен начинаться с **0** или **1**, проверьте первую цифру этих кодов на **0** и **1**. Используйте функцию-элемент потока **clear**, чтобы установить бит **ios::fail**, если ввод неверен.

c) Средняя цифра кода региона всегда **0** или **1**, поэтому проверьте среднюю цифру на **0** и **1**. Используйте функцию-элемент потока **clear**, чтобы установить бит **ios::fail**, если ввод неверен. Если ни одна из вышеупомянутых проверок не установила флаг **ios::fail**, копируйте три части номера телефона в элементы-данные объекта **PhoneNumber** — **areaCode**, **exchange** и **line**. В случае, если бит **ios::fail** установлен, то в основной программе выведите сообщение об ошибке, а не номер телефона.

21.15. Напишите программу, которая выполняет следующие действия:

a) Создайте класс **point**, который содержит элементы целого типа **xCoordinate** и **yCoordinate** и объявляет дружественные функции-операции извлечения из потока и передачи в поток.

b) Определите операцию передачи в поток и операцию извлечения из потока. Функция-операция извлечения из потока должна выполнять проверку введенных данных и, в случае ошибки в данных, устанавливать бит **ios::fail**. Если при вводе значения произошла ошибка, операция передачи не должна выводить объект **point**.

c) Напишите функцию **main**, в которой вводятся и выводятся определяемые пользователем объекты **point** при помощи перегруженных операций извлечения и передачи.

21.16. Напишите программу, которая выполняет следующую задачу:

a) Определите пользовательский класс **complex**, который содержит закрытые целочисленные элементы-данные **real** и **imaginary** и объявляет дружественные классу функции-операции передачи в поток и извлечения из потока.

b) Определите операцию передачи в поток и операцию извлечения из потока. Функция-операция извлечения из потока должна выполнять проверку введенных данных и, в случае ошибки, устанавливать бит **ios::fail**. Вводимые данные должны иметь форму

```
3 + 8i
```

Действительная и мнимая части могут быть отрицательными или положительными, и, кроме того, одно из двух значений может не вводиться. Если значение не введено, соответствующий элемент данных должен быть приравнен **0**. В случае ошибки ввода операция передачи в поток не должна выполнять вывод. Выходной формат должен быть идентичен входному формату, приведенному выше. Для отрицательных мнимых значений вместо знака «плюс» в приведенном выше формате данных должен выводиться знак «минус».

c) Напишите функцию **main**, которая вводит и выводит определяемые пользователем объекты **complex** при помощи перегруженных операций извлечения и передачи.

21.17. Напишите программу, в которой применяется оператор **for** для вывода ASCII-символов со значениями от **33** до **126**. Программа должна вывести десятичное, восьмеричное, шестнадцатеричное значение каждого символа и сам символ. Используйте манипуляторы потока **dec**, **oct** и **hex**.

21.18. Напишите программу, чтобы убедиться, что функции-элементы **istream getline** и **get** с тремя аргументами завершают вводимую строку ограничивающим нуль-символом. Также покажите, что **get** оставляет символ-ограничитель в потоке ввода, в то время как **getline** извлекает ограничитель и отбрасывает его. Что происходит с непрочитанными символами потока?

21.19. Напишите программу, в которой создается определяемый пользователем манипулятор **skipwhite**, пропускающий в потоке ввода пробельные символы. Манипулятор должен использовать функцию **isspace** из библиотеки **ctype.h** для проверки, не является ли символ пробельным. Символы должны вводится функцией-элементом **get** класса **istream**. Как только встречается не-пробельный символ, манипулятор **skipwhite** заканчивает свою работу, помещает этот символ обратно в поток ввода и возвращает ссылку на **istream**.

Протестируйте этот манипулятор в функции **main**, сбросив перед этим флаг **ios::skipws**, чтобы оператор извлечения из потока не пропускал пробельные символы автоматически. Затем проверьте манипулятор, вводя символ, которому предшествует пробельный символ. Выводите извлеченные из потока символы, чтобы убедиться в том, что пробельные символы отбрасываются.

Библиография

(A193) Allison, C., «Code Capsules: A C++ Date Class, Part 1,» *The C Users Journal*, Vol. 11, No. 2, February 1993, pp. 123 – 131.

(An92) Anderson, A. E., and W. J. Heinze, C++ *Programming and Fundamental Concepts*, Englewood Cliffs, NJ: Prentice Hall, 1992.

(Ba93) Bar-David, T., *Object-Oriented Design for* C++, Englewood Cliffs, NJ: Prentice Hall, 1993.

(Be93) Berard, E. V., Essays on Object-Oriented Software Engineering: Volume I, Englewood Cliffs, NJ: Prentice Hall, 1993.

(Br91) Borland, *Borland C++ Programmer's Guide*, Part No. 14MN-TCP04, Scotts Valley, CA: Borland International, Inc., 1991.

(Br91a) Borland, *Borland C++ Getting Started*, Part No. 14MN-TCP02, Scotts Valley, CA: Borland International, Inc., 1991.

(Br91b) Borland, *Borland C++ 3.0 Programmers Guide*, Scotts Valley, CA: Borland International, Inc., 1991.

(Bv93) Byron, D., «The Case for Object Technology Standards,» *CASE Trends*, September 1993, pp. 22–26.

(Co93) Computerworld, «The CW Guide to Object-Oriented Programming,» *Computerworld*, June 14,1993, pp. 107–130.

(De90) Deitel, H. M., *Operating Systems*, Second Edition, Reading, MA: Addison-Wesley, 1990.

(E190) Ellis, M. A. and B. Stroustrup, *The Annotated C++ Reference Manual*, Reading, MA: Addison-Wesley, 1990.

(F193) Flamig, B., *Practical Data Structures in C++*, John Wiley & Sons, 1993.

(Ha93) Hagan, T., «C++ Class Libraries for GUIs,» *Open Systems Today*, Februnry 15, 1993, pp. 54, 58

(Ja93) Jacobson, I., «Is Object Technology Software's Industrial Platform?» *IEEE Software Magazine*, Vol. 10, No. 1, January 1993, pp. 24–30.

(Ko93) Kozaczynski, W., and A. Kuntzmann-Combelles, «What It Takes to Make OO Work,» *IEEE Software Magazine*, Vol. 10, No. 1, January 1993, pp. 20–23.

(Li91) Lippman, S. 8., C++ *Primer* (Second Edition), Reading, MA: Addison-Wesley Publishing Company, 1991.

(Lo93) Lorenz, M., *Object-Oriented Software Development: A Practical Guide*, Englewood Clif'fs. NJ: Prentice Hall, 1993.

(Lu92) Lucas, P. J., *The C++ Programmer's Handbook*, Englewood Cliffs, NJ: Prentice Hall, 1992.

(Ma93) Martin, J., *Principles of Object-Oriented Analysis and Design*, Englewood Cliffs, NJ: Prentice Hall, 1993.

(Me93) Matsche, J. J., «Object-oriented programming in Standard C,» *Object Magazine*, Vol. 2, No. 5, January/February 1993, pp. 71–74.

(Mi91) Microsoft, *Microsoft C/C++ Class Libraries Reference* (Version 7.0), Redmond, WA: Microsoft Corporation, 1991.

(Mi91a) Microsoft, *Microsoft C/C++ C++ Language Reference* (Version 7.0), Redmond, WA: Microsoft Corporation, 1991.

(Mi91b) Microsoft, *Microsoft C/C++ C++ Tutorial* (Version 7.0), Redmond, WA: Microsoft Corporation, 1991.

(Pi93) Pittman, M., «Lessons Learned in Managing Object-Oriented Development,» *IEEE Software Magazine*, Vol. 10, No. 1, January 1993, pp. 43–53.

(Pr93) Prieto-Diaz, R., «Status Report: Software Reusability,» *IEEE Software*, Vol. 10, No. 3, May 1993, pp. 61–66.

(Ra92) Ranade, J., and S. Zamir, C++ *Primer for C Programmers*, New York, NY McGraw-Hill, Inc., 1992.

(Re91) Reed, D. R., «Moving from C to C++,» *Object Magazine*, Vol. 1, No. 3, September/October, 1991, pp. 46–60.

(R193) Rettig, M., G. Simmons, and J. Thomson, «Extended Objects,» *Communications of the ACM*, Vol. 36, No. 8, August 1993, pp. 19–24.

(Sa93) Saks, D., «Inheritance, Part 2,» *The C Users Journal*, May 1993, pp. 81–89.

(Sh91) Shiffman, H., «C++ Object-Oriented Extensions to C,» *SunWorld*, Vol. 4, No. 5, May 1991, pp. 63–70.

(SI693) Skelly. C., «Pointer Power in C and C++, Part 1,» *The C Users Journal*, Vol. 11, No. 2, February 1993, pp. 93–98.

(Sn93) Snyder, A., «The Essence of Objects: Concepts and Terms,» *IEFE Software Magazine*, Vol. 10, No. 1, January 1993, pp. 31–42.

(St91) Stroustrup, B., *The* C++ *Programming Language* (Second Edition), Reading, MA: Addison-Wesley Series in Computer Science, 1991.

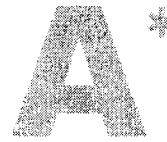
(St93) Stroustrup, B., «Why Consider Language Extensions?: Maintaining a Delicate Balance,» C++ *Report*, September 1993, pp. 44–51.

(Vo93) Voss, G., «Objects and Messages,» *Windows Tech Journal*, February 1993, pp. 15–16.

(Wi93) Wiebel, M., and S. Halladay, «Using OOP Techniques Instead of *switch* in C++,» Vol. 10, No. 10, *The C Users Journal*, October 1992, pp. 105–106.

(WI93) Wilde, N., P. Matthews, and R. Huitt, «Maintaining Object-Oriented Software, *IEEE Software Magazine*, Vol. 10, No. 1, January 1993, pp. 75–80.

(Wt93) Wilt, N., «Templates in C++,» *The C Users Journal*, May 1993, pp. 33–51.



Синтаксис C

В применяемой ниже нотации синтаксические категории (нетерминальные символы) обозначены *курсивом*, а литеральные слова и элементы символьных наборов (терминальные символы) — **полужирным** шрифтом. Двоеточие, следующее за нетерминальным символом, предваряет его определение. Альтернативные определения располагаются на отдельных строках, за исключением случаев, когда им предшествуют слова «один из». Опциональные символы обозначаются подстрочным индексом *opt*, так что

{ *выражение*$_{opt}$ }

означает опциональное выражение, заключенное в фигурные скобки.

Сводка синтаксиса

A.1. Лексическая грамматика

A.1.1. Лексемы

лексема:
    ключевое-слово
    идентификатор
    константа
    строковый-литерал
    операция
    пунктуатор
препроцессорная-лексема:
    имя-заголовка
    идентификатор

---

* Этот материал является сокращением и адаптацией документа Американского Национального Стандарта для Информационных систем — Язык программирования — C, ANSI/ISO 9899; 1990. Копии этого стандарта могут быть получены от Американского Национального Института Стандартов по адресу: 11 West 42nd Street, New York, NY 10036.

pp-число
символьная-константа
строковый-литерал
операция
пунктуатор
любой не-пробельный символ, не являющийся одним из вышеперечисленных

А.1.2. Ключевые слова

ключевое-слово: одно из

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

А.1.3. Идентификаторы

идентификатор:
не-цифра
идентификатор не-цифра.
идентификатор цифра

не-цифра: одна из

```
_ a b c d e f g h i j k l m
  n o p q r s t u v w x y z
  A B C D E F G H I J K L M
  N O P Q R S T U V W X Y Z
```

цифра: одна из

```
0 1 2 3 4 5 6 7 8 9
```

А.1.4. Константы

константа:
плавающая-константа
целая-константа
перечисляемая-константа
символьная-константа

плавающая-константа:
дробная-константа экспоненциальная-часть$_{opt}$ *плавающий-суффикс*$_{opt}$
последовательность-цифр экспоненциальная-часть плавающий-суффикс$_{opt}$

дробная-константа:
последовательность-цифр$_{opt}$. *последовательность-цифр*
последовательность-цифр.

экспоненциальная-часть:
e *знак*$_{opt}$ *последовательность-цифр*
E *знак*$_{opt}$ *последовательность-цифр*

знак: один из
+ -

последовательность-цифр:
цифра
последовательность-цифр цифра

плавающий-суффикс: один из
f l F L

целая-константа:
десятичная-константа целый-суффикс$_{opt}$
восьмеричная-константа целый-суффикс$_{opt}$
шестнадцатеричная-константа целый-суффикс$_{opt}$

десятичная-константа:
ненулевая-цифра
десятичная-константа цифра

восьмеричная-константа:
0
восьмеричная-константа восьмеричная-цифра

шестнадцатеричная-константа:
0x *шестнадцатеричная-цифра*
0X *шестнадцатеричная-цифра*
шестнадцатеричная-константа шестнадцатеричная-цифра

ненулевая-цифра: одна из
1 2 3 4 5 6 7 8 9

восьмеричная-цифра: одна из
0 1 2 3 4 5 6 7

шестнадцатеричная-цифра: одна из
0 1 2 3 4 5 6 7 8 9
a b c d e f
A B C D E F

целый-суффикс:
беззнаковый-суффикс длинный-суффикс$_{opt}$
длинный-суффикс беззнаковый-суффикс$_{opt}$

беззнаковый-суффикс: один из
u U

длинный-суффикс: один из
l L

перечисляемая-константа:
идентификатор

символьная-константа:
'*c-char-последовательность*'
L'*c-char-последовательность*'

c-char-последовательность:
c-char
c-char-последовательность c-char

c-char:
любой элемент набора символов кроме одинарной кавычки ', обратной косой черты \ и символа новой строки
escape-последовательность

escape-последовательность:
простая-escape-последовательность
восьмеричная-escape-последовательность
шестнадцатеричная-escape-последовательность

простая-escape-последовательность: одна из
```
\' \" \? \\
\a \b \f \n \r \t \v
```

восьмеричная-escape-последовательность:
\ *восьмеричная-цифра*
\ *восьмеричная-цифра восьмеричная-цифра*
\ *восьмеричная-цифра восьмеричная-цифра восьмеричная-цифра*

шестнадцатеричная-escape-последовательность:
\x *шестнадцатеричная-цифра*
шестнадцатеричная-escape-последовательность шестнадцатеричная-цифра

A.1.5. Строковые литералы

строковый-литерал:
"*s-char-последовательность*$_{opt}$"
L"*s-char-последовательность*$_{opt}$"

s-char-последовательность:
s-char
s-char-последовательность s-char

s-char:
любой элемент исходного набора символов кроме двойной кавычки ", обратной косой черты \ и символа новой строки
escape-последовательность

A.1.6. Операции

операция: одна из

```
[ ]   ( )   .     ->
++    --    &     *     +     -     ~     !    sizeof
/     %     <<    >>    <     >     <=    >=   ==    !=   ^    |    &&   ||
?     :
=     *=    /=    %=    +=    -=    <<=   >>=  &=    ^=   |=
,     #     ##
```

A.1.7. Пунктуаторы

пунктуатор: один из

```
[ ]   ( )   { }   *     ,     :     =     ;     ...   #
```

A.1.8. Имена заголовков

имя-заголовка:
<h-char-последовательность>
"q-char-последовательность"

h-char-последовательность:
h-char
h-char-последовательность h-char

h-char:
любой элемент исходного набора символов кроме символа новой строки и >

q-char-последовательность:
q-char
q-char-последовательность q-char

q-char:
любой элемент исходного набора символов кроме символа новой строки и "

A.1.9. Препроцессорные числа

pp-число:
цифра
. цифра
pp-число цифра
pp-число не-цифра
pp-число **e** *знак*
pp-число **E** *знак*
pp-число **.**

A.2. Структурная грамматика

A.2.1. Выражения

первичное-выражение:
идентификатор
константа
строковый-литерал
(*выражение* **)**

постфиксное-выражение:
первичное-выражение
постфиксное-выражение **[** *выражение* **]**
постфиксное-выражение **(** *список-выражений-аргументов*$_{opt}$ **)**
постфиксное-выражение **.** *идентификатор*
постфиксное-выражение **->** *идентификатор*
постфиксное-выражение **++**
постфиксное-выражение **--**

список-выражений-аргументов:
присваиваемое-выражение
список-выражений-аргументов **,** *присваиваемое-выражение*

унарное-выражение:
постфиксное-выражение
++ *унарное-выражение*
-- *унарное-выражение*
унарная-операция выражение-приведения
sizeof *унарное-выражение*
sizeof (*имя-типа* **)**

унарная-операция: одна из
& * + - ~ !

выражение-приведения:
унарное-выражение
(*имя-типа* **)** *выражение-приведения*

мультипликативное-выражение:
выражение-приведения
мультипликативное-выражение ***** *выражение-приведения*
мультипликативное-выражение **/** *выражение-приведения*
мультипликативное-выражение **%** *выражение-приведения*

аддитивное-выражение:
мультипликативное-выражение
аддитивное-выражение **+** *мультипликативное-выражение*
аддитивное-выражение **-** *мультипликативное-выражение*

выражение-сдвига:
аддитивное-выражение
выражение-сдвига **<<** *аддитивное-выражение*
выражение-сдвига **>>** *аддитивное-выражение*

выражение-отношения:
выражение-сдвига
выражение-отношения **<** *выражение-сдвига*
выражение-отношения **>** *выражение-сдвига*
выражение-отношения **<=** *выражение-сдвига*
выражение-отношения **>=** *выражение-сдвига*

выражение-равенства:
выражение-отношения
выражение-равенства == выражение-отношения
выражение-равенства != *выражение-отношения*

AND-выражение:
выражение-равенства
AND-выражение & *выражение-равенства*

исключающее-OR-выражение:
AND-выражение
исключающее-OR-выражение ^ *AND-выражение*

включающее-OR-выражение:
исключающее-OR-выражение
включающее-OR-выражение | *исключающее-OR-выражение*

логическое-AND-выражение:
включающее-OR-выражение
логическое-AND-выражение && *включающее-OR-выражение*

логическое-OR-выражение:
логическое-AND-выражение
логическое-OR-выражение || *логическое-AND-выражение*

условное-выражение:
логическое-OR-выражение
логическое-OR-выражение ? *выражение* : *условное-выражение*

присваиваемое-выражение:
условное-выражение
унарное-выражение операция-присваивания присваиваемое-выражение

операция-присваивания: одна из
= *= /= %= += -= <<= >>= ^= |=

выражение:
условное-выражение
выражение , *присваиваемое-выражение*

константное-выражение:
условное-выражение

A.2.2. Объявления

объявление:
спецификаторы-объявления список-инициализирующих-деклараторов$_{opt}$;

спецификаторы-объявления:
спецификатор-класса-хранения спецификаторы-объявления$_{opt}$
спецификатор-типа спецификаторы-объявления$_{opt}$
модификатор-типа спецификаторы-объявления$_{opt}$

список-инициализирующих-деклараторов:
инициализирующий-декларатор
список-инициализирующих-деклараторов , *инициализирующий-декларатор*

инициализирующий-декларатор:
декларатор
декларатор = *инициализатор*

спецификатор-класса-хранения:
typedef
extern
static
auto
register

спецификатор-типа:
void
char
short
int
long
float
double
signed
unsigned
спецификатор-struct-или-union
спецификатор-enum
имя-typedef

спецификатор-struct-или-union:
struct-или-union идентификатор$_{opt}$ { *список-объявлений-struct* }
struct-или-union идентификатор

struct-или-union:
struct
union

список-объявлений-struct:
объявление-struct
список-объявлений-struct объявление-struct

объявление-struct:
список-спецификаторов-и-модификаторов список-деклараторов-struct ;

список-спецификаторов-и-модификаторов:
спецификатор-типа список-спецификаторов-и-квалификаторов$_{opt}$
модификатор-типа список-спецификаторов-и-квалификаторов$_{opt}$

список-деклараторов-struct:
декларатор-struct
список-деклараторов-struct , *декларатор-struct*

декларатор-struct:
декларатор
декларатор$_{opt}$: *константное-выражение*

спецификатор-enum:
enum *идентификатор*$_{opt}$ { *список-перечислителей* }
enum *идентификатор*

список-перечислителей:
перечислитель
список-перечислителей , *перечислитель*

перечислитель:
перечисляемая-константа
перечисляемая-константа = *константное-выражение*

модификатор-типа:
`const`
`volatile`

декларатор:
указатель$_{opt}$ *прямой-декларатор*

прямой-декларатор:
идентификатор
прямой-декларатор [*константное-выражение*]
прямой-декларатор (*список-параметрического-типа*)
прямой-декларатор (*список-идентификаторов*$_{opt}$)

указатель:
* *список-модификаторов-типа*$_{opt}$
* *список-модификаторов-типа*$_{opt}$ *указатель*

список-модификаторов-типа:
модификатор-типа
список-модификаторов-типа модификатор-типа

список-параметрического-типа:
список-параметров
список-параметров , ...

список-параметров:
объявление-параметра
список-параметров , *объявление-параметра*

объявление-параметра:
спецификаторы-объявления декларатор
спецификаторы-объявления абстрактный-декларатор$_{opt}$

список-идентификаторов:
идентификатор
список-идентификаторов , *идентификатор*

имя-типа:
список-спецификаторов-и-квалификаторов абстрактный-декларатор$_{opt}$

абстрактный-декларатор:
указатель
указатель$_{opt}$ *прямой-абстрактный-декларатор*

прямой-абстрактный-декларатор:
(*абстрактный-декларатор* **)**
прямой-абстрактный-декларатор$_{opt}$ **[** *константное-выражение*$_{opt}$ **]**
прямой-абстрактный-декларатор$_{opt}$ **(** *список-параметрического-типа* **)**

имя-typedef:
идентификатор

инициализатор:
присваиваемое-выражение
{ *список-инициализаторов* **}**
{ *список-инициализаторов* **, }**

список-инициализаторов:
инициализатор
список-инициализаторов **,** *инициализатор*

А.2.3. Операторы

оператор:
помеченный-оператор
составной-оператор
оператор-выражение
оператор-выбора
оператор-итерации
оператор-перехода

помеченный-оператор:
идентификатор **:** *оператор*
case *константное-выражение* **:** *оператор*
default **:** *оператор*

составной-оператор:
{ *список-объявлений*$_{opt}$ *список-операторов*$_{opt}$ **}**

список-объявлений:
объявление
список-объявлений объявление

список-операторов:
оператор
список-операторов оператор

оператор-выражение:
выражение$_{opt}$ **;**

оператор-выбора:
if (*выражение*) *оператор*
if (*выражение*) *оператор* **else** *оператор*
switch (*выражение*) *оператор*

оператор-итерации:
while (*выражение*) *оператор*
do *оператор* **while** (*выражение*) ;
for (*выражение*$_{opt}$; *выражение*$_{opt}$; *выражение*$_{opt}$) *оператор*

оператор-перехода:
goto *идентификатор* ;
continue ;
break ;
return *выражение*$_{opt}$;

A.2.4. Внешние определения

модуль-трансляции:
внешнее-объявление
модуль-трансляции внешнее-объявление

внешнее-объявление:
определение-функции
объявление

определение-функции:
спецификаторы-объявления$_{opt}$ *декларатор список-объявлений*$_{opt}$ *составной-оператор*

A.3. Директивы препроцессора

препроцессорный-файл:
группа$_{opt}$

группа:
часть-группы
группа часть-группы

часть-группы:
pp-лексемы$_{opt}$ *новая-строка*
раздел-if
управляющая-строка

раздел-if:
if-группа elif-группы$_{opt}$ *else-группа*$_{opt}$ *endif-строка*

if-группа:
 # if *константное-выражение новая-строка группа*$_{opt}$
 # ifdef *идентификатор новая-строка группа*$_{opt}$
 # ifndef *идентификатор новая-строка группа*$_{opt}$

elif-группы:
elif-группа
elif-группы elif-группа

elif-группа:
 # elif *константное-выражение новая-строка группа*$_{opt}$

else-группа:
 # else *новая-строка группа*$_{opt}$

endif-строка:
 # endif *новая-строка*

управляющая-строка:
 # include *pp-лексемы новая-строка*
 # define *идентификатор список-замены новая-строка*
 # define *идентификатор lparen список-идентификаторов*$_{opt}$ **)**
список-замены новая-строка
 # undef *идентификатор новая-строка*
 # line *pp-лексемы новая-строка*
 # error *pp-лексемы*$_{opt}$ *новая-строка*
 # pragma *pp-лексемы*$_{opt}$ *новая-строка*
*новая-строка*

lparen:
символ левой скобки, которому не предшествует пробельный символ

список-замены:
pp-лексемы$_{opt}$

pp-лексемы:
препроцессорная-лексема
pp-лексемы препроцессорная-лексема

новая-строка:
символ новой строки

Б* Стандартная библиотека

Б.1. Ошибки <errno.h>

EDOM
ERANGE

Расширяются в целочисленные константные выражения с отличающимися ненулевыми значениями, пригодны для использования в директивах препроцессора **#if**.

errno

Переменная типа **int**, значение которой устанавливается равным положительному номеру ошибки различными библиотечными функциями. Значение **errno** равно нулю при запуске программы, но никогда не устанавливается равным нулю какой-либо библиотечной функцией. В программе, использующей **errno** для контроля ошибки, следует установить значение **errno** равным нулю перед вызовом библиотечной функции и проверить ее значение до следующего вызова библиотечной функции. Библиотечная функция может сохранить значение **errno** на входе и установить его равным нулю; исходное значение сохраняется до момента выхода из функции в том случае, если **errno** в это время все еще равна нулю. Ненулевое значение **errno** может быть установлено библиотечной функцией независимо от наличия ошибки, если использование **errno** не документировано описанием этой функции в стандарте.

Б.2. Стандартные определения <stddef.h>

NULL

Зависящая от реализации константа нулевого указателя.

---

* Этот материал является сокращением и адаптацией документа Американского Национального Стандарта для Информационных систем — Язык программирования — C, ANSI/ISO 9899; 1990. Копии этого стандарта могут быть получены от Американского Национального Института Стандартов по адресу: 11 West 42nd Street, New York, NY 10036.

offsetof(*тип*, *обозначение-элемента*)

Расширяется в целочисленное константное выражение типа **size_t**, значение которого представляет смещение в байтах элемента (указанного как *обозначение-элемента*) от начала содержащей его структуры заданного типа (обозначенного как *тип*). *Обозначение-элемента* должно быть таким, что если объявлено

```
static тип t;
```

то значением выражения & (t. *обозначение-элемента*) является адресная константа. (Если специфицированный элемент является битовым полем, то поведение не определено).

ptrdiff_t

Знаковый целочисленный тип результата вычитания двух указателей.

size_t

Беззнаковый целочисленный тип результата операции **sizeof**.

wchar_t

Целочисленный тип, диапазон значений которого может представлять уникальные коды для всех элементов наиболее широкого набора символов, определенного среди поддерживаемых локалов; нулевой символ будет иметь код со значением нуль и каждый элемент базового набора символов будет иметь значение кода, равное его значению при использовании как одиночного символа в целой символьной константе.

Б.3. Диагностика <assert.h>

void assert (int expression)

Макрос **assert** реализует диагностику в программах. Если в момент его исполнения **expression** равно false, макрос **assert** записывает информацию о данном вызове (включая текст аргумента, имя исходного файла и номер строки исходного кода — последние являются также соответственно значениями макросов препроцессора **__FILE__** и **__LINE__**) в стандартный файл ошибок в зависящем от реализации формате. Выведенное сообщение может быть в следующей форме:

```
Assertion failed: expression, file xyz, line nnn
```

Затем макрос **assert** вызывает функцию **abort**. Если в исходном файле, в который включен **assert.h**, имеется директива препроцессора

```
#define NDEBUG
```

то все макросы **assert** в данном файле игнорируются.

Б.4. Обработка символов <ctype.h>

Функции в этом разделе возвращают ненулевые значения (true) в том, и только в том случае, когда значение аргумента **c** соответствует категории, определенной в описании функции.

int isalnum (int c);

Проверка на символ, для которого **isalpha** или **isdigit** равно true.

int isalpha (int c);

Проверка на символ, для которого **isupper** или **islower** равно true.

`int iscntrl (int c);`

Проверка на управляющий символ.

`int isdigit (int c);`

Проверка на десятично-цифровой символ.

`int isgraph (int c);`

Проверка на печатаемый символ, за исключением пробела (' ').

`int islower (int c);`

Проверка на символ, являющийся буквой нижнего регистра.

`int isprint (int c);`

Проверка на печатаемый символ, включая пробел (' ').

`int ispunct (int c);`

Проверка на печатаемый символ, отличный от пробела (' ') и символов, для которых **isalnum** равно true.

`int isspace (int c);`

Проверка на символ, относящийся к стандартным пробельным символам. Стандартными пробельными символами являются: пробел (' '), перевод страницы ('\f'), новая строка ('\n'), возврат каретки ('\r'), символы горизонтальной ('\t') и вертикальной ('\v') табуляции.

`int isupper (int c);`

Проверка на символ, являющийся буквой верхнего регистра.

`int isxdigit (int c);`

Проверка на символ шестнадцатеричной цифры.

`int tolower (int c);`

Преобразование буквы верхнего регистра в нижний. Если аргументом является символ, для которого **isupper** равно true и существует соответствующий символ, для которого **islower** равно true, то функция **tolower** возвращает соответствующий символ; в противном случае аргумент возвращается без изменений.

`int toupper (int c);`

Преобразование буквы нижнего регистра в верхний. Если аргументом является символ, для которого **islower** равно true и существует соответствующий символ, для которого **isupper** равно true, то функция **toupper** возвращает соответствующий символ; в противном случае аргумент возвращается без изменений.

Б.5. Локализация <locale.h>

LC_ALL
LC_COLLATE
LC_CTYPE
LC_MONETARY
LC_NUMERIC
LC_TIME

Расширяются в целочисленные константные выражения с отличающимися значениями, пригодны для использования в качестве первого аргумента функции **setlocale**.

NULL

Зависящая от реализации константа нулевого указателя.

struct lconv

Содержит элементы, связанные с форматированием числовых значений. Структура должна содержать по крайней мере следующие элементы в любом порядке. В локале "C" элементы должны иметь значения, указанные в комментариях.

```
char *decimal_point;        /* "." */
char *thousands_sep;        /* "" */
char *grouping;             /* "" */
char *int_curr_symbol;      /* "" */
char *currency_symbol;      /* "" */
char *mon_decimal_point;    /* "" */
char *mon_thousands_sep;    /* "" */
char *mon_grouping;         /* "" */
char *positive_sign;        /* "" */
char *negative_sign;        /* "" */
char int_frac_digits;       /* CHAR_MAX */
char frac_digits;           /* CHAR_MAX */
char p_cs_precedes;         /* CHAR_MAX */
char p_sep_by_space;        /* CHAR_MAX */
char n_cs_precedes;         /* CHAR_MAX */
char n_sep_by_space;        /* CHAR_MAX */
char p_sign_posn;           /* CHAR_MAX */
char n_sign_posn;           /* CHAR_MAX */
```

char *setlocale (int category, const char *locale);

Функция **setlocale** выбирает часть локала программы, специфицированную аргументами **category** и **locale**. Функция **setlocale** может быть использована для изменения или опроса всего текущего локала программы или его части. Значение **LC_ALL** для **category** обозначает весь локал; другие значения для спецификируют только часть локала программы. **LC_COLLATE** влияет на поведение функций **strcoll** и **strxfrm**. **LC_CTYPE** влияет на поведение функций обработки символов и многобайтовых функций. **LC_MONETARY** влияет на информацию о форматировании денежных значений, возвращаемую функцией **localeconv**. **LC_NUMERIC** влияет на символ десятичной точки для функций форматированного ввода/вывода, функций преобразования строк и информацию о форматировании не-денежных данных, возвращаемую **localconv**. **LC_TIME** влияет на поведение **strftime**.

Значение "C" для **locale** специфицирует минимальные установки для транслятора C; значение "" специфицирует естественные установки для данной реализации. Функции **setlocale** могут быть переданы и другие строки в зависимости от реализации. При запуске программы производятся действия, эквивалентные вызову функции

setlocale (LC_ALL, "C").

Если для **locale** задан указатель на строку и данный выбор может быть удовлетворен, то функция **setlocale** возвращает указатель на строку, ассоциированную с указанным значением **category** для нового локала. Если данный выбор не может быть удовлетворен, то функция **setlocale** возвращает указатель **null** и локал программы не изменяется.

Нулевой указатель для **locale** заставляет функцию **setlocale** возвратить указатель на строку, ассоциированную со значением **category** для текущего локала программы, который при этом не изменяется.

Указатель на строку, возвращаемый функцией **setlocale**, таков, что последующий вызов с этим значением строки и ассоциированной с ним категорией восстановят соответствующую часть локала программы. Строка, на которую ссылается указатель, может модифицироваться программой, но последующий вызов функции **setlocale** может переписать ее.

```
struct lconv *localeconv (void);
```

Функция **localeconv** устанавливает компоненты объекта типа **struct lconv** со значениями, предназначенными для форматирования числовых величин (денежных и других) в соответствии с правилами текущего локала.

Элементы структуры типа **char *** являются указателями на строки, любой из которых (исключая **decimap_point**) может указывать на пустую строку "", показывающую, что значение не доступно в текущем локале или имеет нулевую длину. Элементы типа **char** являются неотрицательными числами, каждое из которых может быть **CHAR_MAX**, показывающую, что значение недоступно в текущем локале. В число элементов входят:

char *decimal_point

Символ «десятичной точки», используемый для форматирования не-денежных величин.

char *thousands_sep

Символ, используемый для отделения групп цифр перед десятичной точкой в форматированных не-денежных величинах.

char *grouping

Строка, элементы которой представляют размер каждой группы цифр в форматированных не-денежных величинах.

char *int_curr_symbol

Международное обозначение валюты, применяемое в текущем локале. Первые три символа строки содержат алфавитные международные обозначения валюты в соответствии со спецификацией ISO 4217:1987. Четвертый символ (непосредственно предшествующий нулевому символу) используется для отделения обозначения валюты от денежной величины.

char *currency_symbol

Обозначение валюты, применяемое в текущем локале.

char *mon_decimal_point

Символ десятичной точки, используемый при форматировании денежных величин.

char *mon_thousands_sep

Разделитель для групп цифр перед десятичной точкой при форматировании денежных величин.

char *mon_grouping

Строка, элементы которой представляют размер каждой группы цифр в форматируемых денежных величинах.

char *positive_sign

Строка, используемая для индикации неотрицательных значений форматируемых денежных величин.

char *negative_sign

Строка, используемая для индикации отрицательных значений форматируемых денежных величин.

char int_frac_digits

Количество цифр в дробной части (после десятичной точки), отображаемых при «международном» форматировании денежных величин.

char frac_digits

Количество цифр в дробной части (после десятичной точки), отображаемых в форматированной денежной величине.

char p_cs_precedes

При установке в 1 или 0 **currency_symbol** *выводится соответственно* впереди или после значения для неотрицательных форматированных денежных величин.

char p_sep_by_space

При установке в 1 или 0 **currency_symbol** соответственно отделяется или не отделяется пробелом от значения для неотрицательных форматированных денежных величин.

char n_cs_precedes

При установке в 1 или 0 **currency_symbol** выводится соответственно впереди или после значения для отрицательных форматированных денежных величин.

char n_sep_by_space

При установке в 1 или 0 **currency_symbol** соответственно отделяется или не отделяется от значения для отрицательных форматированных денежных величин.

char p_sign_posn

Устанавливается равным значению, показывающему позицию **positive_sign** для неотрицательных форматированных денежных величин.

char n_sign_posn

Устанавливается равным значению, показывающему позицию **negative_sign** для отрицательных форматированных денежных величин.

Элементы **grouping** и **mon_grouping** интерпретируются следующим образом:

CHAR_MAX	Далее производить группирование не требуется.
0	Предыдущий элемент должен повторяться для всех оставшихся цифр
другое	Целое значение, равное числу цифр, которое содержит текущая группа. Для определения размера группы цифр перед текущей группой проверяется следующий элемент.

Значения **p_sign_posn** и **n_sign_posn** интерпретируются следующим образом:

0 Величина и **currency_symbol** заключаются в круглые скобки

1 Знаковая строка предшествует величине и **currency_symbol**

2 Знаковая строка следует за величиной и **currency_symbol**

3 Знаковая строка непосредственно предшествует **currency_symbol**

4 Знаковая строка непосредственно следует за **currency_symbol**

Функция **localeconv** возвращает указатель заполненного объекта. Структура, на которую ссылается возвращаемый функцией указатель, не должна модифицироваться программой, но может быть переписана последующим вызовом функции **localeconv**. Кроме того, вызовы функции **setlocale** с категориями **LC_ALL**, **LC_MONETARY** или **LC_NUMERIC** могут переписывать содержимое структуры.

Б.6. Математика <math.h>

HUGE_VAL

Символическая константа, представляющая положительное выражение типа **double**.

double acos(double x);

Вычисление главного значения функции **arccos(x)**. В случае аргументов, не находящихся в диапазоне [-1,+1], происходит ошибка области определения. Функция **acos** возвращает значение арккосинуса в диапазоне $[0,\pi]$ радиан.

double asin(double x);

Вычисление главного значения функции **arcsin(x)**. В случае аргументов, не находящихся в диапазоне [-1,+1], происходит ошибка области определения. Функция asin возвращает значение арксинуса в диапазоне $[-\pi/2,+\pi/2]$ радиан.

double atan(double x);

Вычисление главного значения функции **arctg(x)**. Функция **atan** возвращает значение арктангенса в диапазоне $[-\pi/2,+\pi/2]$ радиан.

double atan2(double y, double x);

Функция **atan2** вычисляет главное значение **arctg(y/x)**, используя знаки обоих аргументов для определения квадранта возвращаемого значения. Ошибка области определения может появиться, если оба аргумента равны нулю. Функция **atan2** возвращает значение арктангенса **y/x** в диапазоне $[-\pi,+\pi]$ радиан.

double cos(double x);

Вычисление **cos(x)** (аргумент в радианах).

double sin(double x);

Вычисление **sin(x)** (аргумент в радианах).

double tan(double x);

Возвращает значение **tg(x)** (аргумент в радианах).

double cosh(double x);

Вычисление гиперболического косинуса **x**. Если величина **x** слишком велика, возникает ошибка диапазона.

double sinh(double x);

Вычисление гиперболического синуса **x**. Если величина **x** слишком велика, возникает ошибка диапазона.

double tanh(double x);

Вычисление гиперболического тангенса **x**.

double exp(double x);

Вычисление экспоненциальной функции от **x**. Если величина **x** слишком велика, возникает ошибка диапазона.

double frexp(double value, int *exp);

Разделение числа с плавающей точкой на нормализованную дробь и целую степень 2. Сохраняет целую часть в объекте типа **int**, указываемом **exp**. Функция **frexp** возвращает значение **x** такое, что **x** является **double** со *значением* в интервале [1/2, 1] или ноль, а **value** равно произведению **x** на 2 в степени ***exp**. Если **value** равно нулю, то обе части результата равны нулю.

double ldexp(double x, int exp);

Произведение числа с плавающей точкой и *целой степени* 2. Может происходить ошибка диапазона. Функция **ldexp** возвращает произведение **x** на 2 в степени **exp**.

double log(double x);

Вычисление натурального логарифма **x**. Если аргумент *является* отрицательным, возникает ошибка области определения. Ошибка диапазона может возникнуть, если аргумент равен нулю.

double log10(double x);

Вычисление десятичного логарифма **x**. Если аргумент *является* отрицательным, возникает ошибка области определения. Ошибка диапазона может возникнуть, если аргумент равен нулю.

double modf(double value, double *iptr);

Разделение аргумента **value** на целую и дробную части, каждая из которых имеет тот же знак, что и аргумент. Сохраняет целую часть как **double** в объекте, на который указывает **iptr**. Функция **modf** возвращает дробную часть **value** со знаком.

double pow(double x, double y);

Вычисление **x** в степени **y**. Ошибка области определения происходит, если **x** является отрицательным, а **y** имеет нецелое значение. Ошибка области определения происходит также, если результат не может быть представлен, когда **x** равно нулю и **y** меньше или равно нулю. Может возникать ошибка диапазона.

double sqrt(double x);

Вычисление неотрицательного квадратного корня из **x**. Если аргумент является отрицательным, происходит ошибка области определения.

double ceil(double x);

Вычисление наименьшего целого значения, не меньшего **x**.

```
double fabs (double x);
```

Вычисление абсолютного значения числа x с плавающей точкой.

```
double floor(double x);
```

Вычисление наибольшего целого значения, не превосходящего x.

```
double fmod(double x, double y);
```

Вычисление остатка от x/y с плавающей точкой.

Б.7. Нелокальные переходы <setjmp.h>

```
jmp_buf
```

Тип массива, используемый для хранения информации, необходимой для восстановления исходного состояния окружения.

```
int setjmp(jmp_buf env);
```

Сохраняет исходное состояние окружения в аргументе **jmp_buf** для позднейшего использования функцией **longjmp**.

Если возврат происходит вследствие непосредственного вызова, то макрос **setjmp** возвращает нулевое значение. Если возврат происходит вследствие вызова функции **longjmp**, то макрос **setjmp** возвращает ненулевое значение.

Активация макроса **setjmp** должна происходить только в одном из следующих контекстов:

- макрос — управляющее выражение оператора выбора или цикла;
- макрос — один из операндов операции сравнения или равенства, с другим операндом — целочисленным константным выражением, причем полученное выражение является управляющим выражением оператора выбора или цикла;
- макрос — операнд унарной операции !, причем полученное выражение является управляющим выражением оператора выбора или цикла; или
- макрос — оператор-выражение.

```
void longjmp(jmp_buf env, int val);
```

Восстановление окружения, сохраненного самым последним вызовом в программе макроса **setjmp** с соответствующим **jmp_buf**-аргументом. Если не было такого вызова или выполнение функции, содержащей вызов макроса **setjmp**, закончилось к данному моменту, то поведение не определено.

Все доступные объекты имеют те же значения, что и во время вызова **longjmp**, за исключением значений объектов с автоматическим периодом хранения, локальных для функции, содержащей вызов соответствующего макроса **setjmp**, не относящихся к типу **volatile** и изменившихся между вызовами **setjmp** и **longjmp**; эти значения являются неопределенными.

Поскольку **longjmp** обходит обычные механизмы вызова и возврата из функции, она будет исполняться корректно в контексте прерываний, сигналов и всех ассоциированных с ними функций. Однако, если функция **longjmp** вызывается из вложенного обработчика сигнала (т.е. из функции, вызванной в результате возбуждения сигнала в процессе обработки другого сигнала), то поведение **longjmp** не определено.

После завершения **longjmp** программа продолжит исполнение, как только соответствующий макрос **setjmp** вернет значение, специфицированное как **val**. Функция **longjmp** не может вынудить макрос **setjmp** вернуть значение 0; если **val** равно нулю, то **setjmp** возвращает значение 1.

Б.8. Обработка сигналов <signal.h>

`sig_atomic_t`

Целочисленный тип объекта, к которому можно обращаться как к атомарной величине даже при наличии асинхронных прерываний.

`SIG_DFL`
`SIG_ERR`
`SEG_IGN`

Расширяются в константные выражения с отличающимися значениями, которые имеют тип, совместимый со вторым аргументом и возвращаемым значением функции **signal**, и чьи значения не могут сравниваться как равные ни с каким адресом объявляемой функции; а также ни с каким из следующих макросов, каждый из которых расширяется в положительное целочисленное константное выражение, являющееся номером сигнала для указанного условия:

SIGABRT	аварийное завершение, аналогичное инициируемому функцией **abort**
SIGFPE	ошибочная арифметическая операция, такая, как деление на ноль или вызывающая переполнение результата
SIGILL	обнаружение недействительного образа функции, такого, как запрещенная инструкция
SIGINT	получение интерактивного сигнала
SIGSEGV	недействительный доступ к памяти
SIGTERM	запрос завершения, посланный программе

Реализация не нуждается в генерировании каких-либо из этих сигналов, за исключением случая явного вызова функции **raise**.

```
void (*signal(int sig, void (*func)(int)))(int);
```

Выбирает один из трех способов обработки полученного впоследствии сигнала с номером **sig**. Если значением **func** является **SIG_DFL**, то для сигнала будет осуществляться обработка по умолчанию. Если значением **func** является **SIG_IGN**, то сигнал будет игнорирован. В других случаях **func** будет указывать на функцию, вызываемую при получении сигнала.Такая функция вызывается *обработчиком сигнала*.

Когда появляется сигнал и **func** указывает на функцию, то сначала выполняется эквивалент вызова **signal(sig, SIG_DFL)**; или производится определяемая реализацией блокировка сигнала. (Если значение **sig** равно **SIGILL**, то реализацией определяется, будет ли выполнена обработка для **SIG_DFL**.) Далее выполняется следующий эквивалент **(*func) (sig)**. Функция **func** может завершиться выполнением оператора **return**, вызовом **abort**, **exit** или **longjmp**. Если **func** выполняет оператор **return** и значение **sig** было **SIGFPE** или какое-либо другое определяемое реализацией значение, соответствующее вычислительным исключениям, то поведение не определено. В противном случае программа будет продолжать исполнение от точки прерывания.

Если сигнал возникает как-либо иначе, чем в результате вызова функции **abort** или **raise**, поведение не определено, если обработчик сигнала вызывает какую-либо функцию стандартной библиотеки помимо самой функции **signal** (с первым аргументом — номером сигнала, соответствующим тому, что вызвал активацию обработчика) или ссылается на какой-нибудь объект со статическим периодом хранения как-либо иначе, чем путем присваивания значения переменной типа **voliatile sig_atomic_t**. Более того, если такой вызов функции **signal** приведет к возврату **SIG_ERR**, то значение **errno** будет неопределенным.

При запуске программы может быть выполнен эквивалент функции

```
signal(sig, SIG_IGN);
```

для некоторых сигналов, выбранных в соответствии с особенностями реализации; для всех других сигналов, определяемых реализацией, выполняется эквивалент функции

```
signal(sig, SIG_DFL);
```

Если запрос может быть удовлетворен, то функция **signal** возвращает значение **func** для самого последнего вызова со специфицированным сигналом **sig**. В других случаях возвращается значение **SIG_ERR** и в **errno** устанавливается положительное значение.

int raise (int sig);

Функция **raise** посылает сигнал **sig** исполняемой программе. Функция **raise** возвращает ноль в случае успешного завершения и ненулевое значение — при неуспешном.

Б.9. Переменные списки аргументов <stdarg.h>

va_list

Тип, пригодный для хранения информации, требуемой макросами **va_start**, **va_arg** и **va_end**. Если нужен доступ к переменному списку аргументов, то вызываемая функция должна объявить объект (обозначаемый в этом разделе как **ap**), имеющий тип **va_list**. Объект **ap** может быть передан как аргумент другой функции; если эта функция вызывает макрос **va_arg** с параметром **ap**, то значение **ap** в вызывающей функции будет неопределенным и должно быть передано макросу **va_end** до любой другой ссылки на **ap**.

void va_start(va_list ap, *parmN***);**

Должна быть вызвана до какого-либо обращения к неименованным аргументам. Макрос **va_start** инициализирует **ap** для последующего использования в **va_arg** и **va_end**. Параметр *parmN* является идентификатором крайнего правого параметра в переменном списке аргументов в определении функции (т.е. первого перед многоточием). Если параметр объявлен с классом памяти **register**, с типом функции или массива, или с типом, не совместимым с тем, что получается после применяемого по умолчанию возведения аргументов, то поведение не определено.

type **va_arg (va_list ap,** *type***);**

Расширяется в выражение, которое имеет тип и значение следующего аргумента в вызове. Параметр **ap** должен быть тем же, что и **va_list**, инициализированный функцией **va_start**. Каждый вызов **va_arg** модифицирует **ap** так, что по очереди возвращаются значения последовательных аргументов. Параметр *type* является именем типа, специфицированного так, что тип указателя на объект, который имеет указанный тип, может быть получен просто применением постфиксной операции * к *type*. Если нет следующего аргумента или *type* не совместим с типом следующего аргумента (после возведения типа, производимого по умолчанию), то поведение не определено. Первый вызов макроса **va_arg** после вызова **va_start** возвратит значение аргумента, следующего за указываемым *parmN*. Последовательные вызовы возвращают значения оставшихся аргументов по порядку.

void va_end (va_list ap);

Упрощает нормальный возврат из функции, переменный список аргументов которой был указан в **va_start**, расширением которого был инициализирован объект **va_list ap**. Макрос **va_end** может модифицировать **ap** так, что его нельзя будет использовать (без промежуточного вызова **va_start**). Если нет соответствующего вызова макроса **va_start**, или перед возвратом не вызван макрос **va_end**, то поведение не определено.

Б.10. Ввод/вывод <stdio.h>

_IOFBF
_IOLBF
_IONBF

Целочисленные константные выражения с отличающимися значениями, пригодные для использования в качестве третьего аргумента функции **setvbuf**.

BUFSIZ

Целочисленное константное выражение, представляющее размер буфера, используемого функцией **setbuf**.

EOF

Отрицательное целочисленное константное выражение, которое возвращается несколькими функциями для указания конца файла, т.е. окончания ввода из потока.

FILE

Тип объекта, способный записывать всю информацию, необходимую для управления потоком, включая индикатор позиции файла, указатель на ассоциированный с ним буфер (если имеется), индикатор ошибки, в который записывается, были или нет ошибки чтения/записи, и индикатор конца файла, регистрирующий, достигнут ли конец файла.

FILENAME_MAX

Целочисленное константное выражение, значение которого представляет размер массива типа **char**, достаточного для хранения наибольшей строки имени файла, который может быть открыт в данной реализации.

FOPEN_MAX

Целочисленное константное выражение, представляющее минимальное число файлов, которое гарантированно может быть одновременно открыто в данной реализации.

fpos_t

Тип объекта, способный записывать всю информацию, необходимую для уникального указания каждой позиции в файле.

L_tmpnam

Целочисленное константное выражение, представляющее необходимый размер для массива типа **char**, достаточного для хранения строки с именем временного файла, генерируемым функцией **tmpnam**.

NULL

Определяемая реализацией константа нулевого указателя.

SEEK_CUR
SEEK_END
SEEK_SET

Целочисленные константные выражения с отличающимися значениями, пригодные для использования в качестве третьего аргумента функции **fseek**.

size_t

Беззнаковый целочисленный тип результата операции **sizeof**.

strderr

Выражение типа «указатель на **FILE**», которое ссылается на объект **FILE**, связанный со стандартным потоком сообщений об ошибках.

stdin

Выражение типа «указатель на **FILE**», которое указывает на объект **FILE**, связанный со стандартным потоком ввода.

stdout

Выражение типа «указатель на **FILE**», которое указывает на объект **FILE**, связанный со стандартным потоком вывода.

TMP_MAX

Целочисленное константное выражение, представляющее минимальное число уникальных имен файлов, которые могут быть генерированы функцией **tmpnam**. Значение макроса **TMP_MAX** должно быть равно, по крайней мере, 25.

int remove(const char *filename);

Делает файл с именем в строке, указанной **filename**, более не доступным под этим именем. Последующая попытка открыть этот файл по данному имени вызовет отказ, если только он не будет создан заново. Если файл открыт, то поведение функции **remove** зависит от реализации. Функция **remove** возвращает ноль, если операция завершена успешно, и ненулевое значение при отказе.

int rename(const char *old, const char *new);

Делает файл, имя которого содержится в строке, указываемой **old**, известным впредь под именем, заданным строкой, указываемой new. Файл с именем **old** является более недоступным под данным именем. Если файл с именем в строке, указываемой **new**, существует до вызова функции **rename**, то ее поведение зависит от реализации. Функция **rename** возвращает ноль, если операция завершена успешно, и ненулевое значение при отказе, в случае чего существующий файл остается под исходным именем.

FILE *tmpfile(void);

Создает временный двоичный файл, который будет автоматически удален при его закрытии или при завершении программы. Если программа завершается аварийно, то останется ли открытый временный файл, зависит от реализации. Файл открывается для модификации в режиме "**wb+**". Функция **tmpfile** возвращает указатель на поток созданного файла. Если файл не может быть создан, то функция **tmpfile** возвращает нулевой указатель.

```
char *tmpnam(char *s);
```

Функция **tmpnam** генерирует строку с допустимым именем файла, которое не совпадает с именем существующего файла. Функция **tmpnam** генерирует различные строки при каждом вызове, до **TMP_MAX** раз. Если число вызовов превысило **TMP_MAX**, то поведение функции зависит от реализации.

Если аргументом является нулевой указатель, то функция **tmpnam** оставляет свой результат во внутреннем статическом объекте и возвращает указатель на этот объект. Последующие вызовы функции **tmpnam** могут модифицировать данный объект. Если аргумент не является нулевым указателем, предполагается, что он указывает на массив символов размером не менее **L_tmpnam**; функция **tmpnam** записывает результат в этот массив и возвращает аргумент в качестве своего значения.

```
int fclose(FILE *stream);
```

Функция **fclose** сбрасывает поток, указываемый параметром **stream**, и закрывает ассоциированный с ним файл. Любые незаписанные буферизованные данные потока передаются системному окружению для записи в файл; все непрочитанные буферизованные данные отбрасываются. Поток отсоединяется от файла. Если ассоциированный буфер был выделен автоматически, то он освобождается. Функция **fclose** возвращает ноль, если поток был успешно закрыт, или **EOF** в случае обнаружения ошибок.

```
int fflush(FILE *stream);
```

Если **stream** указывает на выходной поток или модифицируемый поток, на котором самая последняя операция не была вводом, то функция **fflush** заставляет передать все незаписанные данные этого потока системному окружению для записи в файл; в других случаях поведение не определено.

Если **stream** является нулевым указателем, то функция **flush** выполняет операцию сброса на всех потоках, для которых ее поведение определено выше. Функция **flush** возвращает **EOF**, если произошла ошибка записи, и ноль в противном случае.

```
FILE *fopen(const char *filename, const char *mode);
```

Функция **fopen** открывает файл с именем в строке, указываемой **filename**, и ассоциирует с ним поток. Аргумент **mode** указывает на строку, начинающуюся с одной из следующих последовательностей:

`r`	открыть текстовый файл для чтения
`w`	усечь файл до нулевой длины или создать текстовый файл для записи
`a`	дополнить; открыть или создать текстовый файл для записи данных в конец файла
`rb`	открыть двоичный файл для чтения
`wb`	усечь файл до нулевой длины или создать двоичный файл для записи
`ab`	дополнить; открыть или создать двоичный файл для записи данных в конец файла
`r+`	открыть текстовый файл для обновления (чтение и запись)
`w+`	усечь файл до нулевой длины или создать текстовый файл для обновления
`a+`	дополнить; открыть или создать текстовый файл для обновления, записывая данные в конец

r+b или **rb+** открыть двоичный файл для обновления (чтение и запись)

w+b или **wb+** усечь файл до нулевой длины или создать двоичный файл для обновления

a+b дополнить; открыть или создать двоичный файл для обновления, записывая данные в конец

Открытие файла в режиме чтения ('**r**' как первый символ в аргументе **mode**) вызовет отказ, если файл не существует или не может читаться. Открытие файла в режиме дополнения ('**a**' как первый символ в аргументе **mode**) направляет все последующие записи в файл в текущий конец файла, вне зависимости от возможных вызовов функции **fseek**. В некоторых реализациях открытие двоичного файла в режиме дополнения ('**b**' как второй или третий символ в вышеприведенном списке значений аргумента **mode**) может установить начальное значение индикатора позиции файла для потока за пределами последних записанных данных из-за дополнения нулевыми символами.

Когда файл открыт в режиме обновления ('+' как второй или третий символ в вышеприведенном списке значений аргумента **mode**), то на ассоциированном потоке может производиться как ввод, так и вывод. Однако ввод не может непосредственно следовать за выводом без промежуточного вызова функции **fflush** или функции позиционирования файла (**fseek**, **fsetpos** или **rewind**); вывод также не может непосредственно следовать за вводом без промежуточного вызова функций позиционирования, если только в операции ввода не встретится конец файла. Открытие (или создание) текстового файла в режиме обновления может в некоторых реализациях вместо этого открыть (или создать) двоичный поток.

Когда поток открыт, он полностью буферизуется тогда и только тогда, когда он с определенностью не ссылается на интерактивное устройство. Индикаторы ошибки и конца файла для потока очищаются. Функция **fopen** возвращает указатель на управляющий объект потока. Если при открытии происходит отказ, то **fopen** возвращает нулевой указатель.

```
FILE *freopen(const char *filename, const char *mode,
              FILE *stream);
```

Функция **freopen** открывает файл с именем в строке, на которую указывает **filename**, и ассоциирует с ним поток, на который указывает **stream**. Аргумент mode используется так же, как в функции **fopen**.

Функция **freopen** сначала пытается закрыть любой файл, ассоциированный с указанным потоком. Неудача при закрытии файла игнорируется. Индикаторы ошибки и конца файла для потока очищаются. Функция **freopen** возвращает нулевой указатель, если операция открытия закончилась неудачно. В других случаях **freopen** возвращает значение **stream**.

```
void setbuf(FILE *stream, char *buf);
```

Функция **setbuf** эквивалентна функции **setvbuf**, вызванной со значениями **_IOFBF** для **mode** и **BUFSIZ** для **size** или (если **buf** является нулевым указателем) со значением **_IONBF** для **mode**. Функция **setbuf** ничего не возвращает.

```
int setvbuf(FILE *stream, char *buf, int mode, size_t size);
```

Функция **setvbuf** может быть использована только после того, как поток, на который указывает **stream**, будет ассоциирован с открытым файлом и до выполнения каких-либо других операций с потоком. Аргумент **mode** определяет, как поток будет буферизован: при **_IOFBF** ввод/вывод будет полностью буферизован; при **_IOLBF** ввод/вывод будет буферизован построчно; при **_IONBF** ввод/вывод не буферизуется. Если **buf** не является нулевым указателем, то

вместо буфера, выделенного функцией **setvbuf** , может использоваться массив, на который указывает **buf**. Аргумент **size** специфицирует размер массива. Содержание массива в любой момент является неопределенным. Функция **setvbuf** возвращает ноль при успешном завершении и ненулевое значение, если задано неверное значение для **mode** или запрос не может быть удовлетворен.

```
int fprintf(FILE *stream, const char *format, ...);
```

Функция **fprintf** записывает вывод в поток, указываемый **stream**, под управлением строки, указываемой **format**, которая специфицирует, как последующие аргументы будут преобразованы при выводе. Если для формата недостаточно аргументов, то поведение не определено. Если формат исчерпан, а аргументы остались, то оставшиеся аргументы оцениваются (как обычно), но в остальных отношениях игнорируются. Функция **fprintf** возвращает управление при достижении конца строки формата. См. главу 9, «Форматированный ввод/вывод», где приведены детали спецификации преобразований при выводе. Функция **fprintf** возвращает число переданных символов или отрицательное значение при возникновении ошибки вывода.

```
int fscanf(FILE *stream, const char *format, ...);
```

Функция **fscanf** считывает ввод из потока, на который указывает **stream**, под управлением строки, указываемой **format**, которая специфицирует допустимые входные последовательности и то, как они будут преобразованы для присваивания, используя последующие аргументы как указатели на объекты, получающие преобразованный ввод. Если для формата недостаточно аргументов, то поведение не определено. Если формат исчерпан, а аргументы остались, то оставшиеся аргументы оцениваются (как обычно), но в остальном игнорируются. См. главу 9, «Форматированный ввод/вывод», где приведены детали спецификации преобразований при вводе.

Функция **scanf** возвращает значение макроса **EOF**, если ошибка ввода произошла до каких-либо преобразований. В противном случае функция **fscanf** возвращает количество присвоенных элементов, которое может быть меньше, чем предусматривалось, или даже равно нулю в случае ранней неудачи при сопоставлении формата.

```
int printf(const char *format, ...);
```

Функция **printf** эквивалентна **fprintf** с аргументом **stdout**, помещенным перед аргументами функции **printf**. Функция **printf** возращает количество выведенных символов или отрицательное значение, если произошла ошибка вывода.

```
int scanf(const char *format, ...);
```

Функция scanf эквивалентна функции **fscanf** с аргументом **stdin**, помещенным перед аргументами в **scanf**. Функция **scanf** возвращает значение макроса **EOF**, если при вводе до начала каких-либо преобразований произошла ошибка. В противном случае **scanf** возвращает число введенных элементов, которое может быть меньше, чем предусматривалось, или даже нулю в случае ранней неудачи при сопоставлении формата.

```
int sprintf(char *s, const char *format, ...);
```

Функция **sprintf** эквивалентна **fprintf** за исключением того, что аргумент s специфицирует массив, в который вместо потока будет производиться вывод. В конце записанных символов помещается нулевой символ; он не учитывается при подсчете возвращаемого количества выведенных символов. Поведение при копировании между перекрывающимися объектами не определено. Функция **sprintf** возвращает количество символов, записанных в массив, не считая ограничивающего нулевого символа.

```
int sscanf(const char *s, const char *format, ...);
```

Функция **sscanf** эквивалентна **fscanf** за исключением того, что аргумент **s** специфицирует строку, из которой вместо потока производится ввод. Достижение конца строки эквивалентно появлению конца файла для функции **fscanf**. Поведение при копировании между перекрывающимися объектами не определено.

Функция **sscanf** возвращает значение макроса **EOF**, если при вводе до начала каких-либо преобразований произошла ошибка. В противном случае функция **sscanf** возвращает количество введенных тем, которое может быть меньше, чем предусматривалось, или даже равно нулю в случае ранней ошибки сопоставления.

```
int vfprintf(FILE *stream, const *formar, va_list arg);
```

Функция **vfprintf** эквивалентна **fprintf** с заменой списка аргументов переменной длины аргументом **arg**, который инициализируется макросом **va_start** (возможно, с последующими вызовами **va_arg**). Функция **vfprintf** не вызывает макрос **va_end**. Функция **vfprintf** возвращает количество переданных символов или отрицательное значение, если при выводе произошла ошибка.

```
int vprintf(const char *format, va_list arg);
```

Функция **vprintf** эквивалентна **printf** с заменой списка аргументов переменной длины аргументом **arg**, который должен быть инициализирован макросом **va_start** (возможно, с последующими вызовами **va_arg**). Функция **vprintf** не вызывает макрос **va_end**. Функция **vprintf** возвращает количество переданных символов или отрицательное значение, если при выводе произошла ошибка.

```
int vsprintf(char *s, const char *format, va_list arg);
```

Функция **vsprintf** эквивалентна **sprintf** с заменой списка аргументов переменной длины аргументом **arg**, который должен быть инициализирован макросом **va_start** (возможно, с последующими вызовами **va_arg**). Функция **vsprintf** не вызывает макрос **va_end**. Если имеет место копирование между перекрывающимися объектами, то поведение не определено. Функция **vsprintf** возвращает количество символов, записанных в массив, не считая ограничивающего нулевого символа.

```
int fgetc(FILE *stream);
```

Функция **fgetc** получает следующий символ (если он есть) как **unsigned char**, преобразованный в **int**, из потока ввода с указателем **stream** и перемещает индикатор позиции файла, связанного с потоком (если он определен). Функция **fgetc** возвращает следующий символ из потока ввода, указываемого **stream**. Если поток находится в конце файла, то для потока устанавливается индикатор конца файла и **fgetc** возвращает **EOF**. Если произошла ошибка чтения, то для потока устанавливается индикатор ошибки и **fgetc** возвращает **EOF**.

```
char *fgets(char *s, int n, FILE *stream);
```

Функция **fgets** считывает не более **n** символов из потока, указываемого **stream**, в массив, указываемый **s**. Никаких дополнительных символов после символа новой строки (который не передается) или после конца файла не считывается. Непосредственно за последним символом, считанным в массив, записывается нулевой символ.

Функция **fgets** возвращает **s** в при успешном завершении. Если встретился конец файла и никаких символов в массив не прочитано, то содержимое массива остается без изменений и возвращается нулевой указатель. Если в процессе выполнения произошла ошибка чтения, то содержимое массива является неопределенным и возвращается нулевой указатель.

```
int fputc(int c, FILE *stream);
```

Функция **fputc** записывает символ **c** (преобразованный в **unsigned char**) в поток вывода, указываемый **stream**, в позиции, указываемой индикатором позиции связанного с потоком файла (если определен) и перемещает индикатор. Если файл не может поддерживать запросы позиционирования или в случае, если поток был открыт в режиме дополнения, то символы добавляются в конец потока вывода. Функция **fputc** возвращает записанный символ. Если произошла ошибка записи, то устанавливается индикатор ошибки для потока и **fputc** возвращает **EOF**.

```
int fputs(const char *s, FILE *stream);
```

Функция **fputs** записывает строку, на которую указывает **s**, в поток, указываемый **stream**. Нулевой символ конца строки не записывается. Функция **fputs** возвращает **EOF** в случае ошибки записи и неотрицательное значение в противном случае.

```
int getc(FILE *stream);
```

Функция **getc** эквивалентна **fgetc** за исключением того, что если она реализована как макрос, то может оценивать **stream** более одного раза, поэтому в качестве аргумента следует использовать выражение без побочных эффектов.

Функция **getc** возвращает следующий символ из потока ввода, указываемого **stream**. Если поток находится в конце файла, то для него устанавливается индикатор конца файла и **getc** возвращает **EOF**. В случае ошибки чтения для потока устанавливается индикатор ошибки и **getc** возвращает **EOF**.

```
int getchar(void);
```

Функция **getchar** эквивалентна **getc** с аргументом **stdin**. Функция **getchar** возвращает следующий символ из потока ввода, указываемого **stdin**. Если поток находится в конце файла, то для потока устанавливается индикатор конца файла и **getchar** возвращает **EOF**.

```
char *gets(char *s);
```

Функция **gets** считывает символы из потока ввода, указывемого **stdin**, в массив, на который указывает **s**, до тех пор, пока не будет считан символ новой строки или не встретится конец файла. Символы новой строки отбрасываются и непосредственно после последнего считанного в массив символа записывается нулевой символ. Функция **gets** возвращает **s** в случае успешного завершения. Если встретился конец файла и никаких символов в массив не прочитано, то содержимое массива остается без изменений и возвращается нулевой указатель. Если произошла ошибка чтения в процессе выполнения, то содержимое массива является неопределеннным и возвращается нулевой указатель.

```
int putc(int c, FILE *stream);
```

Функция **putc** эквивалентна **fputc** за исключением того, что в случае ее реализации как макроса возможна неоднократная оценка **stream**, так что аргумент не должен быть выражением с побочными эффектами. Функция **putc** возвращает записанный символ. Если произошла ошибка записи, то для потока устанавливается индикатор ошибки и **putc** возвращает **EOF**.

```
int putchar(int c);
```

Функция **putchar** эквивалентна **putc** со вторым аргументом **stdout**. Функция **putchar** возвращает записанный символ. В случае ошибки записи устанавливается индикатор ошибки для потока и **putchar** возвращает **EOF**.

```
int puts(const char *s);
```

Функция **puts** записывает строку, указываемую **s**, в поток, указываемый **stdout**, и добавляет в конце вывода символ новой строки. Нулевой символ ограничения строки не записывается. Функция **puts** возвращает **EOF**, если произошла ошибка записи; в противном случае она возвращает неотрицательное значение.

```
int ungetc(int c, FILE *stream);
```

Функция **ungetc** возвращает символ **c** (преобразованный в **unsigned char**) обратно в поток ввода, указываемый **stream**. Введенные в поток символы будут возвращены при последующих считываниях из этого потока в обратном порядке. Успешные промежуточные вызовы (для потока, указываемого **stream**) позиционирующих функций (**fseek**, **fstpos** или **rewind**) отбрасывают все введенные в поток символы. Внешняя память, соответствующая потоку, остается без изменений.

Гарантируется возврат в поток одного символа. Если функция **ungetc** вызывается слишком много раз для одного и того же потока без промежуточных операций чтения или позиционирования на нем, функция может потерпеть неудачу. Если значение **c** равно значению **EOF**, операция не удается и поток не изменяется.

Успешный вызов функции **ungetc** очищает индикатор конца файла для потока. Значение индикатора позиции файла для потока после чтения или отбрасывания всех введенных в поток символов будет таким же, как перед возвратом символов в поток. Для текстового потока значение индикатора позиции связанного с ним файла после успешного вызова функции **ungetc** является неспецифицированным до тех пор, пока все введенные в поток символы не будут считаны или отброшены. Для двоичного потока индикатор позиционирования файла определен при каждом удачном вызове функции **ungetc**; если до вызова его значение было равно нулю, то после вызова оно становится неопределеннным. Функция **ungetc** возвращает после преобразования символ, введенный в поток, или **EOF** при неудачной операции.

```
size_t fread(void *ptr, size_t size, size_t nmemb,
             FILE *stream);
```

Функция **fread** считывает в массив, указанный **ptr**, до **nmemb** элементов, размер которых указан параметром **size**, из потока, указываемого **stream**. Индикатор позиции файла для потока (если определен) продвигается вперед на количество успешно прочитанных символов. В случае ошибки конечное значение индикатора позиции файла для потока становится неопределенным. Если элемент считан частично, то его значение не определено.

Функция **fread** возвращает количество успешно считанных элементов, которое может быть меньше, чем **nmemb**, в случае ошибки чтения или появления конца файла. Если **size** или **nmemb** равны нулю, то **fread** возвращает ноль; содержимое массива и состояние остатка потока не изменяются.

```
size_t fwrite(void *ptr, size_t size, size_t nmemb,
              FILE *stream);
```

Функция **fwrite** записывает из массива, указанного **ptr**, до **nmemb** элементов, размер которых указан параметром **size**, в поток, указываемый **stream**. Индикатор позиции файла для потока (если определен) продвигается вперед на количество успешно записанных символов. В случае ошибки конечное значение индикатора позиции файла для потока становится неопределенным. Функция **fwrite** возвращает количество успешно записанных элементов, которое может быть меньше, чем **nmemb**, только в случае ошибки записи.

```
int fgetpos(FILE *stream, fpos_t *pos);
```

Функция **fgetpos** передает текущее значение индикатора позиции файла для потока, указываемого **stream**, в объект, указываемый **pos**. Переданное значение содержит неспецифицированную информацию, используемую функцией **fsetpos** для установления позиции потока в положение, соответствующее моменту вызова функции **fgetpos**. В случае успеха функция **fgetpos** возвращает ноль; при отказе функция **fgetpos** возвращает ненулевое значение и передает определяемое реализацией положительное значение в **errno**.

```
int fseek(FILE *stream, long int offset, int whence);
```

Функция **fseek** устанавливает индикатор позиции файла для потока, указываемого **stream**. Для двоичного потока новая позиция, отсчитанная в символах от начала файла, получается прибавлением **offset** к позиции, задаваемой whence. Заданная позиция соответствует началу файла, если значение whence равно **SEEK_SET**, текущему значению индикатора позиции файла, если **whence** равно **SEEK_CUR**, или концу файла при **whence** равном **SEEK_END**. Для текстового потока значение **offset** должно быть равно либо нулю, либо значению, возвращенному более ранним вызовом функции **ftell** для того же потока, а значение **whence** в последнем случае должно равняться **SEEK_SET**.

Успешный вызов функции **fseek** очищает индикатор конца файла для потока и устраняет все последствия вызовов для него функции **ungetc**. После вызова **fseek** следующей операцией на обновляемом потоке может быть как ввод, так и вывод. Функция **fseek** возвращает ненулевое значение только для запроса, который не может быть удовлетворен.

```
int fsetpos(FILE *stream, const fpos_t *pos);
```

Функция **fsetpos** устанавливает индикатор позиции файла для потока, указываемого **stream**, в соответствии со значением объекта, указываемого **pos**, которое должно быть значением, полученным от более раннего вызова функции **fgetpos** для данного потока. Успешный вызов функции **fsetpos** очищает индикатор конца файла для потока и устраняет все последствия вызовов для него функции **ungetc**. После вызова **fsetpos** следующей операцией на обновляемом потоке может быть как ввод, так и вывод. В случае успеха функция **fsetpos** возвращает ноль; при отказе функция **fsetpos** возвращает ненулевое значение и передает определяемое реализацией положительное значение в **errno**.

```
long int ftell(FILE *stream);
```

Функция **ftell** получает текущее значение индикатора позиции файла для потока, указываемого **stream**. Для двоичных потоков данное значение равно числу символов от начала файла. Для текстового файла его индикатор позиции файла содержит неспецифицированную информацию, используемую функцией fseek для возвращения индикатора позиции файла для потока в положение, соответствующее моменту вызова функции **ftell**; разность между возвращаемыми **ftell** значениями не может быть использована для определения числа записанных или считанных символов. В случае успеха функция **ftell** возвращает текущее значение индикатора позиции файла для потока. При сбое функция **ftell** возвращает **-1L** и передает определяемое реализацией положительное значение в **errno**.

```
void rewind(FILE *stream);
```

Функция **rewind** устанавливает индикатор позиции файла для потока, указанного **stream**, на начало файла. Это эквивалентно

```
(void) fseek (stream, 0L, SEEK_SET)
```

за исключением того, что индикатор ошибки для потока также очищается.

```
void clearerr(FILE *stream);
```

Функция **clearerr** очищает индикаторы конца файла и ошибки для потока, указанного **stream**.

```
int feof(FILE *stream);
```

Функция **feof** проверяет индикатор конца файла для потока, указанного **stream**. Функция **feof** возвращает ненулевое значение тогда и только тогда, когда индикатор конца файла для **stream** установлен.

```
int error(FILE *stream);
```

Функция **ferror** проверяет индикатор ошибки для потока, указываемого **stream**. Функция **ferror** возвращает ненулевое значение в том и только в том случае, если индикатор ошибки для **stream** установлен.

```
void perror(const char *s);
```

Функция **perror** переводит номер ошибки, содержащийся в целом выражении **errno**, в сообщение об ошибке. Она пишет последовательность символов в стандартный поток ошибок следующим образом: сначала (если **s** не является нулевым указателем и символ, на который указывает **s**, не является нулевым) строку, указываемую **s**, двоеточие (:) и пробел; затем соответствующую строку с сообщением об ошибке и символ новой строки. Содержание строки сообщения об ошибке точно такое же, как то, что возвращается функцией **strerror** с аргументом **errno**.

Б.11. Утилиты общего назначения <stdlib.h>

EXIT_FAILURE
EXIT_SUCCES

Целочисленные выражения, которые могут быть использованы в качестве аргумента функции **exit**, возвращающей соответственно неуспешный или успешный статус завершения в системное окружение.

MB_CUR_MAX

Положительное целочисленное выражение, значение которого представляет собой максимальное число байт в многобайтовом символе для набора расширенных символов, установленного в текущем локале (категория **LC_CTYPE**), и которое никогда не превышает **MB_LEN_MAX**.

NULL

Определяемая реализацией константа нулевого указателя.

RAND_MAX

Целочисленное константное выражение, значение которого представляет собой максимальное значение, возвращаемое функцией **rand**. Значение макроса **RAND_MAX** болжно быть равно по крайней мере 32767.

div_t

Тип структуры, представляющей тип значения, возвращаемго функцией **div**.

ldiv_t

Тип структуры, представляющей тип значения, возвращаемого функцией **ldiv**.

size_t

Беззнаковый целый тип результата операции **sizeof**.

wchar_t

Целочисленный тип, диапазон значений которого может представлять уникальные коды для всех элементов наиболее широкого набора символов, определенного среди поддерживаемых локалов; нулевой символ будет иметь код со значением нуль и каждый элемент базового набора символов будет иметь значение кода, равное его значению при использовании как одиночного символа в целой символьной константе.

double atof(const char *nptr);

Преобразует строку, указанную **nptr**, в представление типа **double**. Функция **atof** возвращает преобразованное значение.

int atoi(const char *nptr);

Преобразует строку, указанную **nptr**, в представление типа **int**. Функция **atoi** возвращает преобразованное значение.

long int atol(const char *nptr);

Преобразует строку, указанную **nptr**, в представление типа **long**. Функция **atol** возвращает преобразованное значение.

double strtod(const char *nptr, char **endptr);

Преобразует строку, указанную **nptr**, в представление **double**. Сначала производится декомпозиция строки ввода на три части: начальную, возможно пустую, последовательность пробельных символов (как они определяются функцией **isspace**), предметную последовательность, аналогичную константе с плавающей точкой, и конечную строку из одного или более нераспознанных символов, включая нулевой ограничивающий символ входной строки. Затем функция пытается преобразовать предметную последовательность в число с плавающей точкой и возвратить результат.

Расширенная форма предметной последовательности содержит знаки плюс или минус, затем непустую последовательность цифр, опционально содержащую символ десятичной точки, опциональную экспоненциальную часть, но не «плавающий» суффикс. Предметная последовательность определяется как наибольшая начальная последовательность строки ввода, начинающаяся с первого не-пробельного символа, имеющая ожидаемую форму. Предметная строка не содержит символов, если строка ввода является пустой, содержит исключительно пробельные символы или если первый не-пробельный символ отличен от знака, цифры или десятичной точки.

Если предметная последовательность имеет ожидаемую форму, то последовательность символов, начинающаяся с первой цифры или десятичной точки (с первого любого из данных символов) интерпретируется как константа с плавающей точкой, с тем отличием, что вместо точки используется «символ десятичной точки» и если нет ни экспоненциальной части, ни символа десятичной точки, то десятичная точка предполагается следующей за последней цифрой в строке.

Если предметная последовательность начинается со знака минус, то получаемое после преобразования значение является отрицательным. Указатель на конечную строки передается в объект, указываемый **endptr**, при условии, что **endptr** не является нулевым указателем.

Если предметная последовательность является пустой или не имеет ожидаемой формы, то преобразование не производится; значение **nptr** передается в

объект, указываемый **endptr**, при условии, что **endptr** не является нулевым указателем.

Функция **strtod** возвращает преобразованное значение. Если преобразование не может быть выполнено, то возвращается ноль. Если полученное значение выходит из диапазона представимых значений, то возвращается плюс или минус **HUGE_VAL** (в соответствии со знаком значения) и в **errno** передается значение макроса **ERANGE**.

```
long int strtol(const char *nptr, char **endptr, int base);
```

Преобразует строку, указанную **nptr**, в представление типа **long int**. Сначала производится декомпозиция строки ввода на три части: начальную, возможно пустую, последовательность пробельных символов (как они определяются функцией **isspace**), предметную последовательность, являющуюся допустимым представлением целого в некоторой системе счисления с основанием, определенным значением **base**, и конечную строку с одним или более другими символами, включая нулевой ограничивающий символ входной строки. После этого осуществляется попытка преобразовать предметную последовательность в целочисленное значение и возвратить результат.

Если значение **base** равно нулю, то ожидаемая форма предметной последовательности является такой же, как целая константа со знаком плюс или минус, но не включающая целый суффикс. Если значение **base** находится между 2 и 36, то предполагаемая форма предметной последовательности является последовательностью букв или цифр представления целого с основанием, указанным **base**, с предшествующим знаком плюс или минус, но не включая целый суффикс. Буквам от **a** (или **A**) до **z** (или **Z**) приписываются значения от **10** до 35; допускаются только буквы, чьи значения меньше, чем base. Если значения **base** равно 16, то вслед за знаком, если он имеется, последовательности букв или цифр могут предшествовать символы **0x** или **0X**.

Предметная последовательность, определенная как наибольшая начальная последовательность строки ввода, начинающаяся с первого не-пробельного символа, имеющая ожидаемую форму. Предметная последовательность не содержит символов, если входная строка является пустой или полностью состоит из пробельных символов, или если первый не-пробельный символ отличен от знака или разрешенных букв или цифр.

Если предметная последовательность имеет ожидаемую форму и значение **base** равно нулю, то последовательность символов, начинающаяся с первой цифры, интерпретируется как целая константа. Если предметная последовательность имеет ожидаемую форму и значение **base** находится между 2 и 36, то оно используется как основание для преобразования, а каждой букве приписывается значение, как описано выше. Если предметная последовательность начинается со знака минус, то полученное после преобразования значение является отрицательным. Указатель на конечную строку передается в объект, указываемый **endptr**, при условии, что **endptr** не нуль.

Если предметная последовательность является пустой или не имеет ожидаемой формы, то преобразование не производится; значение **nptr** передается в объект, указываемый **endptr**, при условии, что **endptr** не является нулевым указателем.

Функция **strtol** возвращает преобразованное значение. Если преобразование не может быть выполнено, то возвращается нуль. Если полученное значение выходит из диапазона представимых значений, то возвращается **LONG_MAX** или **LONG_MIN** (в зависимости от знака значения) и в **errno** передается значение макроса **ERANGE**.

unsigned long int strtoul(const char *nptr, char **endptr, int base);

Преобразует строку, указываемую **nptr**, в представление типа **unsigned long int**. Функция **strtoul** работает идентично функции **strtol**. Функция **strtoul** возвращает преобразованное значение. Если преобразование не может быть осуществлено, то возвращается ноль. Если полученное значение находится вне диапазона представимых значений, то возвращается **ULONG_MAX** и в errno передается значение макроса **ERANGE**.

int rand(void);

Функция **rand** вычисляет последовательность псвдослучайных целых чисел в диапазоне от 0 до **RAND_MAX**. Функция **rand** возвращает псевдослучайное целое число.

void srand(unsigned int seed);

Использует аргумент как «семя» для новой последовательности псевдослучайных чисел, возвращаемой последовательными вызовами **rand**. Если **srand** вызывается с одним и тем же семенем, то последовательность псевдослучайных чисел будет повторена. Если **rand** вызывается без какого-либо предварительного вызова **srand**, то будет генерирована такая же последовательность, как в случае вызова **srand** с начальным значением 1. Следующие функции определяют переносимые реализации **rand** и **srand**.

```
static unsigned long int next = 1;

int rand(void) /* RAND_MAX полагается равным 32767 */
{
   next = next * 1103515245 + 12345;
   return (unsigned int) (next/65536) %  32768;
}

void srand (unsigned int seed)
{
   next = seed;
}
```

void *calloc(size_t nmemb, size_t size);

Выделяет пространство для массива из **nmemb** объектов, каждый из которых имеет размер **size**. Выделенная область памяти инициализируется нулевыми байтами. Функция **calloc** возвращает либо нулевой указатель, либо указатель на выделенную область.

void free(void *ptr);

Освобождает область памяти, указанную **ptr**, делая ее доступной для дальнейшего использования. Если **ptr** является нулевым указателем, то никаких действий не происходит. Если аргумент не является указателем, ранее возвращенным функциями **calloc**, **malloc** или **realloc**, или область уже была освобождена вызовом **free** или **realloc**, то поведение не определено.

void *malloc(size_t size);

Выделяет область памяти для объекта с размером, задаваемым **size**. Функция **malloc** возвращает либо нулевой указатель, либо указатель на выделенную область.

```
void *realloc(void *ptr, size_t size);
```

Изменяет размер объекта, указываемого **ptr**, на размер, заданный **size**. Содержимое объекта в диапазоне от начала и вплоть до меньшего из старого и нового размеров сохраняется. Если новый размер больше, то значения во вновь выделенной области объекта являются неопределенными. Если **ptr** является нулевым указателем, то поведение функции **realloc** такое же, как функции **malloc** для указанного размера. Если **ptr** не является указателем, ранее возвращенным функциями **calloc**, **malloc** или **realloc**, или область уже была освобождена вызовом **free** или **realloc**, то поведение не определено. Если область не может быть выделена, то объект, указываемый **ptr**, не изменяется. Если **size** равен нулю и **ptr** не является нулевым указателем, то указываемый объект освобождается. Функция **realloc** возвращает нулевой указатель или указатель на выделенную область, возможно, перемещенную.

```
void abort(void);
```

Производит аварийное завершение программы, если только не перехватывается сигнал **SIGABRT** и обработчик сигнала не возвращает управления. Сбрасываются ли открытые потоки вывода, закрываются ли открытые потоки и удаляются ли временные файлы, определяется реализацией. В системное окружение возвращается зависящая от реализации форма статуса *неуспешного завершения* посредством вызова функции **raise(SIGABRT)**. Функция **abort** не может возвратить управление в вызывающую программу.

```
int atexit(void (*func)(void));
```

Регистрирует функцию, указываемую **func**, для вызова без аргументов при нормальном завершении программы. Реализация должна поддерживать регистрацию по меньшей мере 32-х функций. Функция **atexit** возвращает ноль, если регистрация прошла успешно, и ненулевое значение при отказе.

```
void exit(int status);
```

Вызывает нормальное завершение программы. Если программа исполняет более одного вызова функции **exit**, то поведение не определено. Сначала вызываются все функции, *зарегестрированные функцией* **atexit**, *в порядке, обратном* их регистрации. Каждая функция вызывается столько раз, сколько раз она была зарегистрирована. Далее, все открытые потоки с незаписанными буферизованными данными сбрасываются, все открытые потоки закрываются и все файлы, созданные функцией **tmpfile**, удаляются.

Наконец, управление возвращается в системное окружение. Если значение status равно нулю или **EXIT_SUCCESS**, то возвращается определяемая реализацией форма состояния *успешного завершения*. Если значение **status** равно **EXIT_FAILURE**, то возвращается определяемая реализацией форма состояния *неуспешного завершения*. В остальных случаях возвращаемое состояние зависит от реализации. Функция **exit** не может возвратить управление в вызывающую программу.

```
char *getenv(const char *name);
```

Поиск в *списке окружения*, предусмотренном системным окружением, строки, допускающей сопоставление с той, на которую указывает **name**. Набор имен окружения и метод изменения списка окружения зависят от реализации. Возвращает указатель на строку, ассоциированную с сопоставленным элементом списка. Указываемая строка не должна модифицироваться программой, но может быть переписана последующим вызовом функции **getenv**. Если данное имя не может быть найдено, то возвращается нулевой указатель.

```
int system(const char *string);
```

Передает строку, указываемую **string**, системному окружению для выполнения командным процессором в соответствии с конкретной реализацией. В качестве **string** может быть использован нулевой указатель для выяснения наличия командного процессора. Если аргументом является нулевой указатель, то функция **system** возвращает ненулевое значение только в том случае, если командный процессор доступен. Если аргумент не равен нулевому указателю, то функция **system** возвращает значение, зависящее от реализации.

```
void *bsearch(const void *key, const void *base,
              size_t nmemb, size_t size,
              int (*compar)(const void *, const void *));
```

Поиск в массиве из **nmemb** объектов, начальный элемент которых указан **base**, элемента, соответствующего объекту, указываемому **key**. Размер каждого элемента массива специфицируется аргументом **size**. Функция сравнения, указываемая **compar**, вызывается с двумя аргументами, которые указывают на объект **key** и элемент массива, в данном порядке. Функция должна возвращать целое, меньшее, равное или большее нуля, если рассматриваемый объект **key** соответственно меньше, равен или больше, чем элемент массива. Массив должен состоять из: всех элементов, которые меньше чем, всех элементов, которые равны, и всех элементов, которые больше чем объект **key**, именно в таком порядке.

Функция **bsearch** возвращает указатель соответствующего элемента массива или нулевой указатель, если подходящий элемент не найден. Если два элемента массива эквивалентны, то не специфицируется, который из них будет возвращен.

```
void qsort(void *base, size_t nmemb, size_t size,
           int (*comar)(const void *, const void *));
```

Сортировка массива из **nmemb** объектов. Начальный элемент указывается **base**. Размер каждого объекта специфицируется **size**. Содержимое массива сортируется в восходящем порядке в соответствии с функцией сравнения, указываемой **compar**. Функция должна возвращать целое число, меньшее, равное или большее нуля, если первый аргумент соответственно меньше, равен или больше второго. Если два элемента эквивалентны, то их порядок следования в сортируемом массиве не определен.

```
int abs(int j);
```

Вычисление абсолютного значения целого **j**. Если результат не может быть представлен, то поведение не определено. Функция **abs** возвращает абсолютное значение.

```
div_t div(int numer, int denom);
```

Вычисление частного и остатка от деления числителя **numer** на знаменатель **denom**. Если аргументы не делятся нацело, то полученное частное представляется целым, меньшим, чем ближайшее алгебраическое частное. Если результат не может быть представлен, то поведение не определено; в противном случае выражение ***частное * denom + остаток*** должно быть равно **numer**. Функция **div** возвращает структуру типа **div_t**, включающую как частное, так и остаток. Структура будет содержать следующие элементы в указанном порядке:

```
int quot;          /* частное */
int rem;           /* остаткок */
```

```
long int labs(long int j);
```

Аналогична функции **abs** за исключением того, что аргумент и возвращаемое значение имеют тип **long int**.

```
ldiv_t ldiv(long int numer, long int denom);
```

Аналогична функции **div** за исключением того, что аргументы и элементы возвращаемой структуры (типа **ldiv_t**) имеют тип **long int**.

```
int mblen(const char *s, size_t n);
```

Если **s** не является нулевым указателем, то функция **mblen** определяет число байт, содержащихся в многобайтовом символе, указываемом **s**. Если **s** — нулевой указатель, то функция **mblen** возвращает ненулевое или нулевое значение, если среди кодировок многобайтовых символов соответственно имеются или не имеются кодировки, зависящие от страны. Если **s** не является нулевым указателем, то функция **mblen** возвращает ноль (если **s** указывает на нулевой символ) или количество байт, содержащихся в многобайтовом символе (если следующие **n** или менее байтов образуют допустимый многобайтовый символ), или возвращает -1 (если они не образуют допустимого многобайтового символа).

```
int mbtowc(wchar_t *pwc, const char *s, size_t n);
```

Если **s** не является нулевым указателем, то функция **mbtowc** определяет число байт, которое содержится в многобайтовом символе, указываемом **s**. Она также определяет код для значения типа **wchar_t**, который соответствует этому многобайтовому символу. (Значение кода, соответствующего нулю, равно нулю). Если многобайтовый символ является допустимым и **pwc** не является нулевым указателем, то функция **mbtowc** сохраняет код в объекте, указываемом pwc. Проверяться будут самое большее n байт массива, указываемого **s**.

Если **s** — нулевой указатель, то функция **mblen** возвращает ненулевое или нулевое значение, если среди кодировок многобайтовых символов соответственно имеются или не имеются кодировки, зависящие от страны. Если **s** не является нулевым указателем, то функция **mblen** возвращает ноль (если **s** указывает на нулевой символ) или количество байт, содержащихся в многобайтовом символе (если следующие **n** или менее байтов образуют допустимый многобайтовый символ), или возвращает -1 (если они не образуют допустимого многобайтового символа). Ни в каком случае возвращаемое значение не может быть больше **n** или значения макроса **MB_CUR_MAX**.

```
int wctomb(char *s, wchar_t wchar);
```

Функция **wctomb** определяет число байт, необходимое для представления многобайтового символа, соответствующего коду со значением **wchar** (включая какие-либо изменения регистра). Она сохраняет представление многобайтового символа в массиве, указываемом **s** (если **s** не является нулевым указателем). Сохраняется не более **MB_CUR_MAX** символов. Если значение **wchar** равно нулю, то функция **wctomb** остается в начальном состоянии.

Если **s** — нулевой указатель, то функция **mblen** возвращает ненулевое или нулевое значение, если среди кодировок многобайтовых символов соответственно имеются или не имеются кодировки, зависящие от страны. Если **s** не является нулевым указателем, то функция **wctomb** возвращает -1, если значение **wchar** не соответствует допустимому многобайтовому символу или возвращает число байт, которое содержится в многобайтовом символе, соответствующем значению **wchar**. Ни в каком случае возвращаемое значение не может быть больше значения макроса **MB_CUR_MAX**.

```
size_t mbstowcs(wchar_t *pwcs, const char *s, size_t n);
```

Функция **mbstowcs** преобразует последовательность многобайтовых символов, которая начинается в исходном состоянии регистра, из массива, указываемого **s**, в последовательность соответствующих кодов и сохраняет не более чем **n** кодов в массиве, указываемом **pwcs**. Никакие многобайтовые символы, следующие за нулевым символом (который преобразуется в код со значением ноль) не будут проверяться или преобразовываться. Каждый многобайтовый символ преобразуется так, как если бы вызывалась функция **mbtowc**, за исключением того, что состояние регистра функции **mbtowc** не изменяется.

В массиве, указываемом **pwcs**, будут модифицированы не более **n** элементов. Поведение при копировании между перекрывающимися объектами не определено. Если встречается неправильный многобайтовый символ, то функция **bstowc** возвращает **(size_t) -1**. В противном случае функция **mbstowc** возвращает количество модифицированных элементов массива, не считая нулевого символа завершения строки.

```
size_t wcstombs(char *s, const wchar_t *pwcs, size_t n);
```

Функция **wcstombs** преобразует последовательность кодов, которые соответствуют многобайтовым символам, из массива, указываемого **pwcs**, в последовательность многобайтовых символов, которая начинается в исходном состоянии регистра, и сохраняет эти многобайтовые символы в массиве, указываемом **s**, останавливаясь, если многобайтовый символ будет превышать по размеру **n** байтов или если сохраняется нулевой символ. Каждый код преобразуется, как если бы вызывалась функция **wctomb**, за исключением того, что состояние регистра функции **wctomb** не изменяется.

В массиве, указываемом **s**, будет модифицировано не более **n** байтов. Если имеет место копирование между перекрывающимися объектами, то поведение не определено. Если встречается код, который не соответствует допустимому многобайтовому символу, то функция **wcstombs** возвращает **(size_t) -1**. В противном случае функция **wcstombs** возвращает число модифицированных байтов, не считая нулевого символа завершения строки.

Б.12. Обработка строк <string.h>

NULL

Определяемая реализацией константа нулевого указателя.

size_t

Беззнаковый целый тип результата операции **sizeof**.

```
void *memcpy(void *s1, const void *s2, size_t n);
```

Функция **memcpy** копирует **n** символов из объекта, указываемого **s2**, в объект, указываемый **s1**. Если копирование имеет место между объектами, которые перекрываются, то поведение не определено. Функция **memcpy** возвращает значение **s1**.

```
void *memmove(void *s1, const void *s2, size_t n);
```

Функция **memmove** копирует **n** символов из объекта, указываемого **s2**, в объект, указываемый **s1**. Копирование происходит так, как если бы **n** символов из объекта, указанного **s2**, сначала копировались во временный массив из **n** символов, который не перекрывается с объектами, на которые указывают **s1** и **s2**, и затем **n** символов из временного массива копировались бы в объект, указываемый **s1**. Функция **memmove** возвращает значение **s1**.

```
char *strcpy(char *s1, const char *s2);
```

Функция **strcpy** копирует строку, указываемую **s2** (включая нулевой символ ограничения строки) в массив, указываемый **s1**. Если копирование имеет место между перекрывающимися объектами, то поведение не определено. Функция **strcpy** возвращает значение **s1**.

```
char *strncpy(char *s1, const char *s2, size_t n);
```

Функция **strncpy** копирует не более **n** символов (символы, которые следуют за нулевым символом, не копируются) из массива, указываемого **s2**, в массив, указываемый **s1**. Если копирование имеет место между перекрывающимися объектами, то поведение не определено. Если массив, указываемый **s2**, является строкой, которая содержит менее **n** символов, то к их копии в массив, указываемый **s1**, добавляется соответствующее количество нулевых символов. Функция **strncpy** возвращает значение **s1**.

```
char *strcat(char *s1, const char *s2);
```

Функция **strcat** присоединяет копию строки, указываемой **s2** (включая ограничивающий нулевой символ), в конец строки, указываемой **s1**. Начальный символ **s2** переписывает нулевой символ в конце **s1**. Если копирование имеет место между перекрывающимися объектами, то поведение не определено. Функция **strcat** возвращает значение **s1**.

```
char *strncat(char *s1, const char *s2, size_t n);
```

Функция **strncat** присоединяет не более **n** символов (нулевой символ и символы, следующие за ним, не копируются) из массива, указываемого **s2**, в конец строки, указываемой **s1**. К результату всегда присоединяется ограничивающий нулевой символ. Если копирование имеет место между перекрывающимися объектами, то поведение не определено. Функция **strncat** возвращает значение **s1**.

```
int memcmp(const void *s1, const void *s2, size_t n);
```

Функция **memcmp** сравнивает первые n символов объекта, указываемого **s1**, с первыми **n** символами объекта, указываемого **s2**. Функция **memcmp** возвращает целое большее, равное или меньшее нуля, если объект, указываемый **s1**, соответственно больше, равен или меньше объекта, указанного **s2**.

```
int strcmp(const char *s1, const char *s2);
```

Функция **strcmp** сравнивает строку, указываемую **s1**, со строкой, указываемой **s2**. Функция **strcmp** возвращает целое большее, равное или меньшее нуля, если строка, указываемая **s1**, соответственно больше, равна или меньше строки, указываемой **s2**.

```
int strcoll(const char *s1, const char *s2);
```

Функция **strcoll** сравнивает строку, указываемую **s1**, со строкой, указываемой **s2**, причем обе интерпретируются как соответствующие категории **LC_LOCAL** текущего локала. Функция **strcoll** возвращает целое большее, равное или меньшее нуля, если строка, указываемая **s1**, соответственно больше, равна или меньше чем строка, указываемая **s2**, если обе строки интерпретировать в текущем локале.

```
int strncmp(const char *s1, const char s2, size_t n);
```

Функция **strncmp** сравнивает не более чем **n** символов (символы, следующие за нулевым символом, не сравниваются) из массива, указываемого **s1**, с символами из массива, указываемого **s2**. Функция **strncmp** возвращает целое большее, равное или меньшее нуля, если массив (возможно завершающийся нулем), указываемый **s1**, соответственно больше, равен или меньше массива (возможно завершающегося нулем), указываемого **s2**.

```
size_t strxfrm(char *s1, const char s2, size_t n);
```

Функция **strxfrm** преобразует строку, указывемую **s2**, и помещает преобразованную строку в массив, указываемый **s1**. Преобразование таково, что если к двум преобразованным строкам применяется функция **strcmp**, то возвращаемое ею значение будет больше, равно или меньше нуля в соответствии с результатом применения функции **strcoll** к двум исходным строкам. В массив результата, указываемый **s1**, записывается не более **n** символов, включая ограничивающий нулевой символ. Если **n** равно нулю, то **s1** может быть нулевым указателем. Если имеет место копирование перекрывающихся объектов, то поведение не определено. Функция **strxcmp** возвращает длину преобразованной строки (не считая конечного нулевого символа). Если значение равно или больше **n**, то содержимое массива, указываемого **s1**, является неопределенным.

```
void *memchr(const void *s, int c, size_t n);
```

Функция **memchr** находит первое появление аргумента **c** (преобразованного в **unsigned char**) в начальных **n** символах (каждый из которых интерпретируется как **unsigned char**) объекта, указываемого **s**. Функция **memchr** возвращает указатель на найденный символ или нулевой указатель, если символ в объекте не встретился.

```
char *strchr(const char *s, int c);
```

Функция **strchr** находит первое вхождение параметра **c** (преобразованного в **char**) в строку, указываемую **s**. Завершающий нулевой символ рассматривается как входящий в строку. Функция **strchr** возвращает указатель на найденный символ или нулевой указатель, если символ в строке не встретился.

```
size_t strcspn(const char *s1, const char *s2);
```

Функция **strcspn** вычисляет длину максимального начального сегмента строки, указываемой **s1**, который полностью состоит из символов, не входящих в строку, указываемую **s2**. Функция **strcspn** возвращает длину данного сегмента.

```
char *strpbrk(const char *s1, const char *s2);
```

Функция **strpbrk** находит первое вхождение в строку, указываемую **s1**, какого-либо символа из строки, указываемой **s2**. Функция **strpbrk** возвращает указатель на символ или нулевой указатель, если нет вхождений символов из **s2** в **s1**.

```
char *strrchr(const char *s, int c);
```

Функция **strrchr** находит последнее вхождение параметра **c** (преобразованного в **char**) в строку, указываемую **s**. Завершающий нулевой символ считается входящим в строку. Функция **strrcch** возвращает указатель на символ или нулевой указатель, если **c** в строке не обнаружен.

```
size_t strspn(const char *s1, const char *s2);
```

Функция **strspn** вычисляет длину максимального начального сегмента строки, указываемой **s1**, который полностью состоит из символов строки, указанной **s2**. Функция **strspn** возвращает длину сегмента.

```
char *strstr(const char *s1, const char *s2);
```

Функция **strstr** находит первое появление в строке, указываемой **s1**, последовательности символов (исключая завершающий нулевой символ) строки, указываемой **s2**. Функция **strstr** возвращает указатель на найденную строку или нулевой указатель, если строка не найдена. Если **s2** указывает на строку нулевой длины, то функция возвращает **s1**.

```
char *strtok(char *s1, const char *s2);
```

Последовательность вызовов функции **strtok** разбивает строку, указываемую **s1**, на последовательность лексем, которые отделены друг от друга символами из строки, указанной **s2**. Первый вызов в последовательности использует как аргумент **s1**; последующие вызовы осуществляются с нулевым указателем в качестве первого аргумента. Разделяющая строка, указанная **s2**, может отличаться от вызова к вызову.

Первый вызов осуществялет поиск в строке, указываемой **s1**, первого символа, который не содержится в текущей строке разделителей, указываемой **s2**. Если таких символов не найдено, то в этом случае в строке, указываемой **s1**, нет лексем и функция **strtok** возвращает нулевой указатель. Если такой элемент найден, то он будет началом первой лексемы.

Затем функция **strtok** ищет символ, который содержится в текущей разделяющей строке. Если такой символ не найден, то текущая лексема простирается до конца строки, указываемой **s1**, и следующий поиск лексемы возвратит нулевой указатель. Если такой символ найден, то он заменяется нулевым символом, который завершает текущую лесему. Функция **strtok** сохраняет указатель на следующий символ, с которого начнется поиск следующей лексемы.

Каждый последовательный вызов с нулевым указателем как значением первого аргумента начинает поиск от сохраненного указателя аналогично описанному выше. Функция **strtok** возвращает указатель на первый символ лексемы или нулевой указатель, если лексемы отсутствуют.

```
void *memset(void *s, int c, size_t n);
```

Функция **memset** копирует значение **c** (преобразованное в **unsigned char**) в каждый из первых **n** символов объекта, указываемого **s**. Функция **memset** возвращает значение **s**.

```
char *strerror(int errnum);
```

Функция **strerror** ставит в соответствие номеру ошибки в **errnum** строку сообщения об ошибке. Функция **strerror** возвращает указатель на строку, содержимое которой определяется реализацией. Указываемый массив не должен модифицироваться программой, но может быть переписан последующим вызовом функции **strerror**.

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

Функция **strlen** вычисляет длину строки, указываемой **s**. Функция **strlen** возвращает количество символов, которое предшествует ограничивающему строку нулевому символу.

Б.13. Дата и время <time.h>

CLOCKS_PER_SEC

Число, соответствующее одной секунде в значении, возвращаемом функцией **clock**.

NULL

Определяемая реализацией константа нулевого указателя.

clock_t

Арифметический тип, способный представлять время.

time_t

Арифметический тип, способный представлять время.

size_t

Беззнаковый целый тип результата операции **sizeof**.

struct tm

Содержит компоненты календарного времени, называемого *разделенным временем*. Структура должна содержать по крайней мере следующие элементы в любом порядке. Семантика элементов и их нормальный диапазон значений представлены в комментариях.

```
int tm_sec;          /* секунды после минут - [0,61] */
int tm_min;          /* минуты после часов  - [0,59] */
int tm_hour;         /* часы с полуночи - [0,23] */
int tm_mday;         /* день месяца - [1,31] */
int tm_mon;          /* месяц с января - [0,11] */
int tm_year;         /* год с 1900 */
int tm_wday;         /* день с воскресенья - [0,6] */
int tm_yday;         /* день с января - [0,365] */
int tm _isdst;       /* флаг перехода на летнее время */
```

Значение **tm_isdst** является положительным, если переход на летнее время имеет место, равно нулю, если его не происходит, и отрицательным, если об этом нет информации.

clock_t clock(void);

Функция **clock** определяет использованное процессорное время. Функция **clock** возвращает наилучшее приближение процессорного времени, использованного программой, от начала определяемой реализацией эры, относящейся только к активации программы. Для определения времени в секундах значение, возвращаемое функцией **clock**, должно быть поделено на значение макроса **CLOCKS_PER_SEC**. Если использованное процессорное время недоступно или его значение не может быть представлено как **clock_t**, то функция возвращает значение **(clock_t) -1**.

double difftime(time_t time1, time_t time0);

Функция **difftime** вычисляет разность между двумя календарными моментами времени: **time1 - time0**. Функция **difftime** возвращает разность как **double**, выраженную в секундах.

time_t mktime(struct tm *timeptr);

Функция **mktime** преобразует разделенное время, выраженное как местное, в структуре, указываемой **timeptr**, в значение календарного времени с такой же кодировкой, как значение, возвращаемое функцией **time**. Исходные значения компонентов **tm_wday** и **tm_yday** структуры игнорируются, а значения других компонентов не ограничиваются диапазонами, указанными выше. При успешном завершении значения компонентов **tm_wday** и **tm_yday** структуры устанавливаются соответствующим образом и другие компоненты устанавливаются представляющими специфицированное календарное время, но со значениями в диапазонах, указанных выше; окончательное значение **tm_mday** не устанавливается до тех пор, пока не определятся значения **tm_mon** и **tm_year**. Функция **mktime** возвращает специфицированное календарное время, кодированное как значение типа **time_t**. Если данное календарное время не может быть представлено как **time_t**, то функция возвращает значение **(time_t) -1**.

```
time_t time(time_t *timer);
```

Функция **time** определяет текущее календарное время. Функция **time** возвращает наилучшее представление текущего календарного времени в данной реализации. В случае, если календарное время является недоступным, возвращается значение **(time_t) -1**. Если **timer** не является нулевым указателем, то возвращаемое значение присваивается также объекту, на который указывает **timer**.

```
char *asctime(const struct tm *timeptr);
```

Функция **asctime** преобразует разделенное время из структуры, указываемой **timeptr**, в строку следующего вида:

```
Sun Sep 16 01:03:52 1973\n\0
```

Функция **asctime** возвращает указатель на строку.

```
char *ctime(const time_t *timer);
```

Функция **ctime** преобразует календарное время, указываемое **timer**, в местное время в виде строки. Это эквивалентно следующему вызову:

```
asctime(localtime(timer))
```

Функция **ctime** возвращает указатель, возвращаемый функцией **asctime** с таким разделенным временем как аргументом.

```
struct tm *gmtime(const time_t *timer);
```

Функция **gmtime** преобразует календарное время, указываемое **timer**, в разделенное время, выраженное как UTC — Coordinated Universal Time. Функция **gmtime** возвращает указатель на этот объект или нулевой указатель, если UTC недоступно.

```
struct tm *localtime(const time_t *timer);
```

Функция **localtime** преобразует календарное время, указываемое **timer**, в разделенное время, выраженное как местное. Функция **localtime** возвращает указатель на этот объект.

```
size_t strtime(char *s, size_t maxsize, const char *format,
               const struct tm *timeptr);
```

Функция **strtime** помещает символы в массив, указываемый **s**, в соответствии с управляющей строкой **format**. Строка **format** содержит нуль или более спецификаторов преобразования и ординарные многобайтовые символы. Все ординарные символы (включая нулевой символ ограничения строки) копируются без изменения в массив. Если имеет место копирование между перекрывающимися объектами, то поведение не определено. В массиве размещается не более **maxsize** символов. Каждый спецификатор преобразования замещается соответствующими символами, как описано в приведенном ниже списке. Эти символы определяются категорией **LC_TIME** текущего локала и значениями, содержащимися в структуре, указываемой **timeptr**.

- **%a** заменяется локальным сокращенным обозначением дня недели.
- **%A** заменяется локальным полным названием дня недели.
- **%b** заменяется локальным сокращенным обозначением месяца.
- **%B** заменяется локальным полным названием месяца.
- **%c** заменяется локальным представлением соответствующей даты и времени.
- **%d** заменяется днем месяца как десятичным числом **(01-31)**.

- **%H** заменяется часом (24-часовые часы) как десятичным числом (**00-23**).
- **%I** заменяется часом (12-часовые часы) как десятичным числом (**01-12**).
- **%j** заменяется днем года как десятичным числом (**001-366**).
- **%m** заменяется месяцом как десятичным числом (**01-12**).
- **%M** заменяется минутой как десятичным числом (**00-59**).
- **%p** заменяется локальным обозначением AM/PM, связанным с 12-часовыми часами.
- **%S** заменяется секундой как десятичным числом (**00-61**).
- **%U** заменяется десятичным номером недели года (первое воскресенье как первый день недели 1) (**00-53**).
- **%w** заменяется днем недели как десятичным числом (**0-6**), где воскресенью соответствует номер **0**.
- **%W** заменяется десятичным номером недели года (первый понедельник как первый день недели 1) (**00-53**).
- **%x** заменяется локальным представлением даты.
- **%X** заменяется локальным представлением времени.
- **%y** заменяется годом века, как десятичным числом (**00-99**).
- **%Y** заменяется годом с веком как десятичным числом.
- **%Z** заменяется названием часового пояса или остается пустым, если часовой пояс не определен.
- **%%** заменяется %.

Если спецификатор преобразования не является одним из указанных выше, то поведение не определено. Если общее число символов результата, включая нулевой символ ограничения строки, не превышает **maxsize**, то функция **strtime** возвращает число символов, записанных в массив, указываемый **s**, не считая нулевого символа окончания строки. В противном случае возвращается ноль и содержание массива является неопределенным.

Б.14. Ограничения реализации

<limits.h>

Следующие макросы должны быть определены со значениями не меньшими (по абсолютной величине) указанных ниже.

#define CHART_BIT 8

Число бит для наименьшего объекта, который не является битовым полем (байт).

#define SCHAR_MIN -127

Минимальное значение для объекта типа **signed char**.

#define SCHAR_MAX +127

Максимальное значение для объекта типа **signed char**.

#define UCHAR_MAX 255

Максимальное значение для объекта типа **unsigned char**.

#define CHAR_MIN **0 или CHAR_MIN**

Минимальное значение для объекта типа **char**.

#define CHAR_MAX **UCHAR_MAX или SCHAR_MAX**

Максимальное значение для объекта типа **char**.

#define MB_LEN_MAX **1**

Максимальное число байт в многобайтовом символе для любого из поддерживаемых локалов.

#define CHRT_MIN **-32767**

Минимальное значение для объекта типа **short int**.

#define SHRT_MAX **+32767**

Максимальное значение для объекта типа **short int**.

#define USHRT_MAX **65535**

Максимальное значение для объекта типа **unsigned short int**.

#define INT_MIN **-32767**

Минимальное значение для объекта типа **int**.

#define INT_MAX **+32767**

Максимальное значение для объекта типа **int**.

#define UINT_MAX **65535**

Максимальное значение для объекта типа **unsigned int**.

#define LONG_MIN **-2147483647**

Минимальное значение для объекта типа **long int**.

#define LONG_MAX **+2147483647**

Максимальное значение для объекта типа **long int**

#define ULONG_MAX **4294967295**

Максимальное значение для объекта типа **unsigned long int**

<float.h>

#define FLT_ROUNDS

Режим округления при сложении чисел с плавающей точкой.

-1 неопределенный

0 к нулю

1 к ближайшему

2 к положительной бесконечности

3 к отрицательной бесконечности

Следующие макросы должны быть определены со значениями не меньшими (по абсолютной величине) указанных ниже.

```
#define FLT_RADIX                                    2
```

Основание экспоненциального представления, *b*

```
#define FLT_MANT_DIG
#define LDBL_MANT_DIG
#define DBL_MANT_DIG
```

Количество цифр (в системе с основанием **FLT_RADIX**) для чисел с плавающей точкой, *p*.

```
#define FLT_DIG                                      6
#define DBL_DIG                                     10
#define LDBL_DIG                                    10
```

Количество десятичных цифр, *q*, такое, что любое число с плавающей точкой, содержащее *q* десятичных цифр, может быть округлено до числа с плавающей точкой по основанию *b*, содержащего *p* цифр, и преобразовано обратно без изменения *q* десятичных цифр.

```
#define FLT_MIN_EXP
#define DBL_MIN_EXP
#define LDBL_MIN_EXP
```

Минимальное отрицательное целое, такое, что число **FLT_RADIX**, возведенное в эту степень минус 1, является нормализованным числом с плавающей точкой.

```
#define FLT_MIN_10_EXP                             -37
#define DBL_MIN_10_EXP                             -37
#define LDBL_MIN_10_EXP                            -37
```

Минимальное отрицательное целое, такое, что 10 в данной степени находится в допустимом диапазоне нормализованных чисел с плавающей точкой.

```
#define FLT_MAX_EXP
#define DBL_MAX_EXP
#define LDBL_MAX_EXP
```

Максимальное целое, такое, что **FLT_RADIX**, возведенное в эту степень минус 1, является представимым конечным числом с плавающей точкой.

```
#define FLT_MAX_10_EXP                             +37
#define DBL_MAX_10_EXP                             +37
#define LDBL_MAX_10_EXP                            +37
```

Максимальное целое, такое, что 10 в данной степени находится в допустимом диапазоне представимых конечных чисел с плавающей точкой.

Следующие макросы должны быть определены со значениями не меньшими указанных ниже.

```
#define FLT_MAX                                  1E+37
#define DBL_MAX                                  1E+37
#define LDBL_MAX                                 1E+37
```

Максимальное представимое конечное число с плавающей точкой.

Следующие макросы должны быть определены со значениями не большими указанных ниже.

```
#define FLT_EPSILON                    1E-5
#define DBL_EPSILON                    1E-9
#define LDBL_EPSILON                   1E-9
```

Разность между 1.0 и наименьшим значением, большим 1.0, которая представима в данном типе с плавающей точкой.

```
#define FLT_MIN                        1E-37
#define DBL_MIN                        1E-37
#define LDBL_MIN                       1E-37
```

Минимальное нормализованное положительное число с плавающей точкой.



Таблица приоритета операций

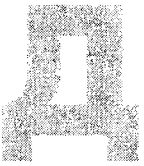
Операции	Ассоциативность
() [] -> .	слева направо
++ -- + - ! ~ (тип) * & sizeof	справа налево
* / %	слева направо
+ -	слева направо
<< >>	слева направо
< <= > >=	слева направо
== !=	слева направо
&	слева направо
^	слева направо
|	слева направо
&&	слева направо
||	слева направо
?:	справа налево
= += -= *= /= %= &= ^= |= <<= >>=	справа налево
,	слева направо

Операции расположены в порядке убывания приоритета от верхних к нижним.

Набор символов ASCII

	0	1	2	3	4	5	6	7	8	9
0	nul	soh	stx	etx	eot	enq	ack	bel	bs	ht
1	nl	vt	ff	cr	so	si	dle	dc1	dc2	dc3
2	dc4	nak	syn	etb	can	em	sub	esc	fs	gs
3	rs	us	sp	!	"	#	$	%	&	'
4	(	)	*	+	,	-	.	/	0	1
5	2	3	4	5	6	7	8	9	:	;
6	<	=	>	?	@	A	B	C	D	E
7	F	G	H	I	J	K	L	M	N	O
8	P	Q	R	S	T	U	V	W	X	Y
9	Z	[	\	]	^	_	`	a	b	c
10	d	e	f	g	h	i	j	k	l	m
11	n	o	p	q	r	s	t	u	v	w
12	x	y	z	{	|	}	~	del		

Цифры слева от таблицы являются первыми цифрами десятичного эквивалента кода символа (0–127), а цифры вверху таблицы представляют собой последнюю цифру кода символа. Например, код символа 'F' равен 70, а код символа '&' — 38.



Системы счисления



Цели

- Усвоить основные понятия систем счисления, такие, как основание системы счисления, позиционное значение и числовое значение.
- Понять методику работы с числами, представленными в различных системах счисления: двоичной, восьмеричной и шестнадцатеричной.
- Научиться записывать двоичные числа в виде восьмеричных и шестнадцатеричных чисел.
- Научиться преобразовывать восьмеричные и шестнадцатеричные числа к двоичному виду.
- Научиться выполнять прямое и обратное преобразование десятичных чисел и их двоичных, восьмеричных и шестнадцатеричных эквивалентов.
- Понять двоичную арифметику и представление отрицательных двоичных чисел с помощью метода дополнения до двух.

Содержание

Д.1. Введение

Д.2. Представление двоичных чисел как восьмеричных и шестнадцатеричных

Д.3. Преобразование восьмеричных и шестнадцатеричных чисел в двоичные

Д.4. Преобразование двоичных, восьмеричных или шестнадцатеричных чисел в десятичные

Д.5. Перевод десятичных чисел в двоичные, восьмеричные или шестнадцатеричные

Д.6. Отрицательные двоичные числа: дополнение до двух

Резюме • Упражнения для самоконтроля • Ответы на упражнения для самоконтроля • Упражнения

Д.1. Введение

В данном приложении мы представим основные системы счисления, которые используют в программировании на языке С, особенно в тех случаях, когда разрабатываются программы, требующие взаимодействия с оборудованием на «машинном» уровне. К таким программам относятся операционные системы, сетевое программное обеспечение, компиляторы, системы баз данных, а также приложения, от которых требуется высокая эффективность исполнения.

Когда мы пишем целое, например, 227 или -63, то используем *десятичную (по основанию 10) систему счисления*. Исходными цифрами в десятичной системе являются 0, 1, 2, 3, 4, 5, 6, 7, 8, 9. Наименьшей из них является 0, наибольшей 9, которая на единицу меньше, чем *основание* 10. «Внутри себя» компьютер использует *двоичную (по основанию 2) систему счисления*. Двоичная система счисления содержит только две исходных цифры, а именно 0 и 1. Наименьшей является 0, наибольшей 1 — на единицу меньше, чем основание 2.

Мы увидим, что двоичные числа, как правило, значительно длиннее их десятичных эквивалентов. Программисты, работающие на языке ассемблера или на языке высокого уровня, подобном С, предоставляющем возможность

«опуститься» на машинный уровень программирования, находят работу с двоичными числами весьма обременительной. Поэтому две другие системы счисления — *восьмеричная* (по основанию 8) и *шестнадцатеричная* (по основанию 16) — являются очень популярными, поскольку делают удобной запись двоичных чисел.

В восьмеричной системе исходные цифры изменяются в диапазоне от 0 до 7. Поскольку как двоичная, так и восьмеричная системы счисления имеют меньше цифр, чем десятичная, то их цифры такие же, как и в десятичной системе.

В шестнадцатеричной системе возникает проблема записи чисел, поскольку здесь требуется шестнадцать цифр — наименьшая из которых 0 и наибольшая со значением, эквивалентным десятичному числу 15 (на единицу меньше основания 16). Для представления шестнадцатеричных чисел со значениями от 10 до 15 (в десятичном исчислении) условились использовать буквы от A до F. Так, в шестнадцатеричном представлении мы можем написать, например, число 876, имеющее десятично-подобный вид и содержащее только цифры; число 8A55F, содержащее цифры и буквы, и, например, число FFE, состоящее только из букв. Порой шестнадцатеричное число можно произносить по буквам, подобно словам, например FACE или FEED — это может показаться странным программисту, привыкшему работать с десятичными числами.

Двоичные цифры	Восьмеричные цифры	Десятичные цифры	Шестнадцатеричные цифры
0	0	0	0
1	1	1	1
	2	2	2
	3	3	3
	4	4	4
	5	5	5
	6	6	6
	7	7	7
		8	8
		9	9
			A (десятичное значение 10)
			B (десятичное значение 11)
			C (десятичное значение 12)
			D (десятичное значение 13)
			E (десятичное значение 14)
			F (десятичное значение 15)

Рис. Д.1. Цифры двоичной, восьмеричной, десятичной и шестнадцатеричной систем счисления

В любой из указанных систем счисления используется *позиционная нотация* — каждое место, на котором стоит цифра, имеет различное *позиционное значение*. Например, в десятичном числе 937 (цифры 9, 3 и 7 называют *символическими значениями*), мы говорим, что 7 записана в *позиции единиц*, 3 — в *позиции десятков*, 9 — в *позиции сотен*. Отметим, что каждая из этих позиций представляет собой соответствующую степень основания 10; эти степени начинаются с 0 и увеличиваются на 1 по мере перемещения влево вдоль числа (рис. Д.3).

Атрибут	Двоичная система	Восьмеричная система	Десятичная система	Шестнадцатеричная система
Основание	2	8	10	16
Наименьшая цифра	0	0	0	0
Наибольшая цифра	1	7	9	F

Рис. Д.2. Сравнение двоичной, восьмеричной, десятичной и шестнадцатеричной систем счисления

Позиционные значения в десятичной системе счисления			
Десятичная цифра	9	3	7
Имя позиции	Сотен	Десятков	Единиц
Позиционное значение	100	10	1
Позиционное значение как степень основания (10)	10^2	10^1	10^0

Рис. Д.3. Позиционные значения в десятичной системе счисления

Для больших десятичных чисел следующими позициями слева будут *позиции тысяч* (10 в 3-й степени), *десятков тысяч* (10 в 4-й степени), *сотен тысяч* (10 в 5-й степени), *миллионов* (10 в 6-й степени), *десятков миллионов* (10 в 7-й степени) и т.д.

В двоичном числе 101 мы видим, что крайняя правая единица записана в *позиции единиц*, 0 находится в *позиции двух* и крайняя левая единица в *позиции четырех*. Заметим, что каждая из этих позиций представляет собой степень основания (числа 2); эти степени начинаются с 0, увеличиваясь на единицу по мере смещения по числу влево (рис. Д.4).

Для больших двоичных чисел следующими позициями слева будут *позиции восьми* (2 в 3-й степени), *шестнадцати* (2 в 4-й степени), *тридцати двух* (2 в 5-й степени), *шестидесяти четырех* (2 в 6-й степени) и т.д.

Для восьмеричного числа 425 мы говорим, что 5 записана в *позиции единиц*, 2 — в *позиции восьми* и 4 в *позиции шестидесяти четырех*. Заметим, что каждая из этих позиций представляет степень основания (числа 8); эти степени начинаются с 0, увеличиваясь на единицу по мере смещения по числу влево (рис. Д.5).

Для больших восьмеричных чисел следующими слева будут *позиции пятисот двенадцати* (8 в 3-й степени), *четырех тысяч девяноста шести* (8 в 4-й), *тридцати двух тысяч семисот шестидесяти восьми* (8 в 5-й степени) и т.д.

Позиционные значения в двоичной системе счисления			
Двоичная цифра	1	0	1
Имя позиции	Четырех	Двух	Единиц
Позиционное значение	4	2	1
Позиционное значение как степень основания (2)	2^2	2^1	2^0

Рис. Д.4. Позиционные значения в двоичной системе счисления

Позиционные значения в восьмеричной системе счисления			
Восьмеричная цифра	4	2	5
Имя позиции	Шестидесяти четырех	Восьми	Единиц
Позиционное значение	64	8	1
Позиционное значение как степень основания (8)	8^2	8^1	8^0

Рис. Д.5. Позиционные значения в восьмеричной системе счисления

Для шестнадцатеричного числа 3DA мы говорим, что A находится в *позиции единиц*, D — в *позиции шестнадцати* и 3 — в *позиции двухсот пятидесяти шести*. Заметим также, что каждая из этих позиций представляет степень основания системы счисления (числа 16); эти степени начинаются с 0, увеличиваясь на единицу по мере смещения по числу влево (рис. Д.6).

Для длинных шестнадцатеричных чисел следующими слева будут *позиции четырех тысяч девяноста шести* (16 в степени 3), *шестидесяти пяти тысяч пятисот тридцати шести* (16 в 4-й степени) и т.д.

Позиционные значения в шестнадцатеричной системе счисления			
Шестнадцатеричная цифра	3	D	A
Имя позиции	Двухсот пятидесяти шести	Шестнадцати	Единиц
Позиционное значение	256	16	1
Позиционное значение как степень основания (16)	16^2	16^1	16^0

Рис. Д.6. Позиционные значения в шестнадцатеричной системе счисления

Д.2. Запись двоичных чисел в виде восьмеричных и шестнадцатеричных

Основное применение восьмеричных и шестнадцатеричных чисел в компьютерных вычислениях заключается в сокращении длины записи двоичного представления чисел. Рис. Д.7 наглядно демонстрирует, как большие

двоичные числа можно записать в очень краткой, сжатой форме в системах счисления с большим основанием, чем двоичная система.

Десятичное число	Двоичное представление	Восьмеричное представление	Шестнадцатеричное представление
0	0	0	0
1	1	1	1
2	10	2	2
3	11	3	3
4	100	4	4
5	101	5	5
6	110	6	6
7	111	7	7
8	1000	10	8
9	1001	11	9
10	1010	12	A
11	1011	13	B
12	1100	14	C
13	1101	15	D
14	1110	16	E
15	1111	17	F
16	1000	20	10

Рис. Д.7. Десятичные, двоичные, восьмеричные и шестнадцатеричные эквиваленты

Весьма важным соотношением между этими системами является то, что как восьмеричная, так и шестнадцатеричная системы счисления имеют основания (соответственно 8 и 16), являющиеся степенями основания двоичной системы (числа 2). Рассмотрим следующее двенадцатизначное двоичное число и его восьмеричный и шестнадцатеричный эквиваленты. Попробуйте, если удастся, установить взаимосвязь между двоичной записью числа и его восьмеричным и шестнадцатеричным эквивалентами. Ответ на это дают следующие числа.

```
Двоичное          Восьмеричный      Шестнадцатеричный
число             эквивалент        эквивалент
100011010001      4321              8D1
```

Чтобы понять, как можно легко преобразовать двоичное число в восьмеричное, просто разделите приведенное выше 12-значное двоичное число на группы по три бита в каждой и затем запишите эти группы над соответствующими цифрами восьмеричного эквивалента, следующим образом:

```
100     011     010     001
4       3       2       1
```

Заметьте, что восьмеричные цифры, записанные под каждой группой из трех бит, точно соответствуют восьмеричным эквивалентам трехзначных двоичных чисел, приведенным на рис. Д.7.

Подобным же образом можно заметить закономерность преобразования числа из двоичной системы в шестнадцатеричную. А именно, разделите 12-значное двоичное число на группы по четыре бита в каждой и затем запишите эти группы над соответствующими цифрами шестнадцатеричного эквивалента, как это сделано ниже:

```
1000    1101    0001
8       D       1
```

Опять же заметьте, что шестнадцатеричные цифры, записанные под каждой группой из четырех бит, точно соответствуют шестнадцатеричным эквивалентам 4-значных двоичных чисел, приведенным на рис. Д.7.

Д.3. Преобразование восьмеричных и шестнадцатеричных чисел в двоичные

В предыдущем разделе мы видели, как преобразовать двоичные числа в их восьмеричные или шестнадцатеричные эквиваленты путем формирования групп из двоичных цифр и перезаписи каждой из групп с помощью соответствующего эквивалентного восьмеричного или шестнадцатеричного значения. Такой же подход можно использовать в обратном порядке при нахождении двоичного эквивалента для данного восьмеричного или шестнадцатеричного числа.

Например, восьмеричное число 653 переводится в двоичное путем простой записи 6 как его трехзначного двоичного эквивалента 110, 5 — как 3-значного двоичного эквивалента 101, 3 — как 011, так что в целом число записывается в виде 9-значного двоичного числа 110101011.

Шестнадцатеричное число FAD5 преобразуется в двоичное с помощью записи F как 4-значного двоичного эквивалента 1111, числа A как двоичного 1010, числа D как 1101 и 5 как 0101, что дает 16-значное число 1111101011010101.

Д.4. Преобразование из двоичной, восьмеричной или шестнадцатеричной форм числа в десятичную

Поскольку мы привыкли работать в десятичной системе счисления, то очень часто приходится преобразовывать двоичные, восьмеричные или шестнадцатеричные числа в десятичные для того, чтобы почувствовать его действительное значение. Наши диаграммы в разделе Д.1 выражают позиционные значения в десятичной системе. Для преобразования числа из какой-либо системы в десятичную необходимо умножить каждую цифру числа на десятичный эквивалент ее позиционного значения и все результаты просуммировать. Для примера, двоичное число 110101 преобразуется в десятичное число 53 так, как показано на рис. Д.8.

Для преобразования восьмеричного числа 7614 в соответствующее десятичное 3980 используется аналогичная методика с применением в этом случае восьмеричных позиционных значений, как показано на рис. Д.9.

Преобразование двоичного числа в десятичное						
Позиционные значения:	32	16	8	4	2	1
Символические значения:	1	1	0	1	0	1
Произведения:	1*32=32	1*16=16	0*8=8	1*4=4	0*2=0	1*1=1
Сумма:	=32+16+8+4+0+1=53					

Рис. Д.8. Преобразование двоичного числа в десятичное

Преобразование восьмеричного числа в десятичное				
Позиционные значения:	512	64	8	1
Символические значения:	7	6	1	4
Произведения:	7*512=3584	6*64=384	1*8=8	4*1=4
Сумма:	= 3584 + 384 + 8 + 4 = 3980			

Рис. Д.9. Преобразование восьмеричного числа в десятичное

Преобразование шестнадцатеричного числа AD3B в десятичное 44347 осуществляется аналогично с применением в этом случае шестнадцатеричных позиционных значений, как показано на рис. Д.10.

Преобразование шестнадцатеричного числа в десятичное				
Позиционные значения:	4096	256	16	1
Символические значения:	A	D	3	B
Произведения:	A*4096=40960	D*256=3328	3*16=48	B*1=11
Сумма:	= 40960 + 3328 + 48 + 11 = 44347			

Рис. Д.10. Преобразование шестнадцатеричного числа в десятичное

Д.5. Преобразование десятичного числа в двоичное, восьмеричное или шестнадцатеричное

Преобразования в предыдущих разделах естественным образом вытекали из соответствующих соглашений о позиционной записи чисел. Способы преобразования из десятичного представления числа в двоичное, восьмеричное или шестнадцатеричное также вытекают из данных положений.

Предположим, что мы хотим перевести десятичное число 57 в двоичное. Мы начнем с записи в ряд позиционных значений справа налево в порядке их возрастания. Ненужные члены ряда отбросим. Итак, сначала запишем:

```
Позиционные значения:   64   32   16   8   4   2   1
```

Отбросим член ряда с позиционным значением 64 :

```
Позиционные значения:        32   16   8   4   2   1
```

Затем начнем работать слева направо. Разделив 57 на 32 видим, что в 57 укладывается одно значение 32 с остатком 25, поэтому запишем 1 в столбец с 32. Далее, делим 25 на 16; в 25 укладывается одно значение 16 и остаток 9, поэтому записываем 1 в столбец с 16. Делим остаток 9 на 8, получаем единицу, которую запишем в столбец с 8. При делении остатка 1 на позиционные значения в следующих двух позициях получаем нули, которые и записываем в столбцы соответствующие 4 и 2. И, наконец, деление 1 на 1 даст единицу, которую пишем в столбце с 1. В результате:

```
Позиционные значения:        32   16   8   4   2   1
Символические значения:      1    1    1   0   0   1
```

Таким образом, десятичному числу 57 соответствует двоичное число 111001.

Для преобразования десятичного числа 103 в восьмеричное мы также начнем с записи позиционных значений в ряд до тех пор, пока не получим позиционное значение, превышающее данное десятичное число. Нам не потребуется последний член ряда, поэтому отбросим его. Таким образом, сначала запишем:

```
Позиционные значения:   512   64   8   1
```

Затем, отбрасывая член с позиционным значением 512, получаем:

```
Позиционные значения:        64   8   1
```

Далее работаем со столбцами слева направо. Делим 103 на 64, получаем единицу и 39 в остатке, поэтому записываем 1 в столбец с 64. Делим остаток 39 на 8, получаем 4 и 7 в остатке; записываем 4 в столбец с 8. Наконец, деля остаток 7 на 1, получаем значение 7, которое записываем в столбец с 1. В результате:

```
Позиционные значения:        64   8   1
Символические значения:      1    4   7
```

Таким образом, десятичному числу 103 соответствует восьмеричное 147.

Для преобразования десятичного числа 375 в шестнадцатеричное мы также начнем с записи позиционных значений в ряд, пока не получим позиционное значение, превышающее данное десятичное число. Нам не потребуется последний член ряда, поэтому отбросим его. Таким образом, сначала запишем:

```
Позиционные значения:   4096   256   16   1
```

Затем, отбрасывая член с позиционным значением 4096, получаем

```
Позиционные значения:        256   16   1
```

Далее работаем слева направо. Делим 375 на 256, получаем единицу и 119 в остатке, поэтому записываем 1 в столбец с 256. Делим остаток 119 на 16, получаем 7 и 7 в остатке; записываем 7 в столбец с 16. Наконец, деля остаток 7 на 1, получаем значение 7, которое записываем в столбец с 1. В результате:

```
Позиционные значения:        256   16   1
Символические значения:      1     7    7
```

Таким образом, десятичному числу 375 соответствует шестнадцатеричное 177.

Д.6. Отрицательные двоичные числа: дополнение до двух

Обсуждение в данном приложении до сих пор было сфокусировано на работе с положительными числами. В этом разделе мы объясним, как в компьютерах представляются отрицательные числа с помощью *нотации дополнения до двух*. Сначала поясним, как формируется указанное дополнение до двух и затем покажем, почему оно представляет отрицательное значение данного двоичного числа.

Предположим, что имеем машину с 32-битным представлением целых чисел. Пусть в программе объявлено:

```
int value = 13;
```

Тогда 32-битовое представление **value** будет иметь вид

```
00000000 00000000 00000000 00001101
```

Для представления отрицательного значения **value** мы сначала используем применяемую в языке C операцию поразрядного дополнения (~):

```
ones_complement_of_value = ~value;
```

Во внутреннем представлении **~value** представляет собой **value**, в котором каждый бит заменен на противоположное значение: единицы заменены нулями, а нули — единицами:

```
value:
00000000 00000000 00000000 00001101

~value (т.е. дополнение value до единицы):
11111111 11111111 11111111 11110010
```

Для образования дополнения **value** до двух мы просто прибавим единицу к дополнению его до единицы. В этом случае имеем:

Дополнение **value** до двух

```
11111111 11111111 11111111 11110011
```

Если это число действительно равно -13, то, прибавив к нему 13, мы должны получить 0. Попробуем проделать это:

```
 00000000 00000000 00000000 00001101
+11111111 11111111 11111111 11110011
-----------------------------------
 00000000 00000000 00000000 00000000
```

Бит переноса из крайнего левого разряда отбрасывается, так что мы действительно в результате получили ноль. Если мы прибавим дополнение числа до единицы к самому числу, то в результате получим 1 во всех битах. Ключ к получению результата со всеми нулями в том, что дополнение до двух на 1 больше, чем дополнение до единицы. Поэтому прибавление 1 привело к тому, что в каждой позиции получился ноль с переносом единицы. Этот перенос прошел по всему числу до крайней левой позиции и был за ней отброшен, что в результате дало все нули.

Разность двух чисел

```
x = a - value;
```

представляется в компьютере с использованием дополнения **value** до двух следующим образом:

```
x = a + (~value + 1).
```

Предположим, что число **a** равно 27 и **value**, как и ранее, равно 13. Если дополнение **value** до двух действительно представляет отрицательное значение **value**, то по последней формуле мы должны получить в результате 14. Давайте проделаем эти операции:

```
a (т.е. 27)        00000000 00000000 00000000 00011011
+ (~value + 1)    +11111111 11111111 11111111 11110011
                  -----------------------------------
                   00000000 00000000 00000000 00001110
```

в результате чего мы действительно получаем 14.

Резюме

- Когда мы в программе на языке C записываем целое, такое, как 19, 227 или -63, то по умолчанию предполагается, что число представлено в десятичной (по основанию 10) системе счисления. Цифрами в десятичной системе счисления являются 0, 1, 2, 3, 4, 5, 6, 7, 8 и 9. Наименьшей цифрой является 0, наибольшей 9, которая на единицу меньше основания 10.
- Для внутреннего представления компьютер использует двоичную (по основанию 2) систему счисления. Двоичная система счисления имеет только две цифры, а именно 0 и 1. Наименьшей цифрой является 0, наибольшей 1, что на единицу меньше, чем основание 2.
- Восьмеричная (по основанию 8) и шестнадцатеричная (по основанию 16) системы счисления являются весьма распространенными прежде всего потому, что дают удобную форму короткой записи двоичных чисел.
- Цифры в восьмеричной системе счисления изменяются в диапазоне от 0 до 7.
- В шестнадцатеричной системе счисления имеется проблема обозначений, поскольку требуется шестнадцать цифр — наименьшая 0 и наибольшая со значением, эквивалентным десятичному 15 (на единицу меньше, чем основание системы 16). По соглашению для представления шестнадцатеричных цифр используются буквы от A до F, соответствующие десятичным значениям от 10 до 15.
- Каждая система счисления использует позиционную запись — каждая позиция, в которой записывается цифра, имеет различное позиционное значение.
- Весьма важным является то, что восьмеричная и шестнадцатеричная системы счисления имеют основания (соответственно 8 и 16), представляющие собой степени 2, т.е. основания двоичной системы счисления.
- Для перевода восьмеричного числа в двоичное необходимо просто заменить каждую восьмеричную цифру на трехзначный двоичный эквивалент.
- Для перевода шестнадцатеричного числа в двоичное необходимо просто заменить каждую шестнадцатеричную цифру на четырехзначный двоичный эквивалент.

- Поскольку мы привыкли работать в десятичной системе счисления, то часто бывает удобным перевести двоичное, восьмеричное или шестнадцатеричное число в десятичное для того, чтобы лучше почувствовать, что это за число «на самом деле».
- Для преобразования числа из какой-либо системы в десятичную необходимо умножить каждую цифру числа на десятичный эквивалент ее позиционного значения и все результаты просуммировать.
- Компьютеры представляют отрицательные числа с использованием нотации дополнения до двух.
- Чтобы получить в двоичной системе отрицательное значение числа, сначала образуется его дополнение до единицы посредством применения операции C для поразрядного дополнения (поразрядного НЕ, ~). Тем самым биты числа оказываются обращенными (1 вместо 0 и наоборот). Образование дополнения числа до двух осуществляется простым прибавлением единицы к дополнению его до единицы.

Терминология

восьмеричная система счисления
двоичная система счисления
десятичная система счисления
метод дополнения до единицы
метод дополнения до двух
основание системы счисления
отрицательная величина
позиционная запись
позиционное значение
поразрядная операция дополнения (~)
преобразование числа из одной системы счисления в другую
система счисления по основанию 10
система счисления по основанию 16
система счисления по основанию 2
система счисления по основанию 8
цифра
числовое значение
шестнадцатеричная система счисления

Упражнения для самоконтроля

Д.1. Основаниями десятичной, двоичной, восьмеричной и шестнадцатеричной систем счисления являются соответственно _____, _____, _____ и _____.

Д.2. В общем случае десятичное, восьмеричное и шестнадцатеричное представление данного двоичного числа будет содержать (больше/меньше) цифр, чем двоичное представление.

Д.3. (Верно/неверно) Одной из главных причин использования десятичной системы счисления является то, что эта форма удобна для сокращенной записи двоичных чисел путем простой замены десятичной цифры на группу из четырех двоичных бит.

Д.4. (Восьмеричное/шестнадцатеричное/десятичное) представление большого двоичного числа является наиболее кратким.

Д.5. (Верно/неверно) Наибольшая цифра любой системы счисления на единицу больше ее основания.

Д.6. (Верно/неверно) Наименьшая цифра любой системы счисления на единицу меньше ее основания.

Д.7. Позиционное значение для крайней правой цифры любой системы счисления — двоичной, восьмеричной, десятичной или шестнадцатеричной — всегда равно _____.

Д.8. Позиционное значение слева от крайней правой цифры любого числа в двоичной, восьмеричной, десятичной или шестнадцатеричной системах всегда равно _____.

Д.9. Заполните пропуски в таблице позиционных значений для данных систем счисления:

десятичная	1000	100	10	1
шестнадцатеричная	...	256	...	...
двоичная	...	...	...	...
восьмеричная	512	...	...	...

Д.10. Преобразуйте двоичное число **110101011000** в восьмеричное и шестнадцатеричное представления.

Д.11. Преобразуйте шестнадцатеричное число **FACE** в двоичное.

Д.12. Преобразуйте восьмеричное число **7316** в двоичное.

Д.13. Преобразуйте шестнадцатеричное число **4FEC** в восьмеричное. (Подсказка: сначала переведите **4FEC** в двоичное число, а затем из двоичного — в восьмеричное).

Д.14. Преобразуйте двоичное число **1101110** в десятичное.

Д.15. Преобразуйте восьмеричное число **317** в десятичное.

Д.16. Преобразуйте шестнадцатеричное число **EFD4** в десятичное.

Д.17. Преобразуйте десятичное число **177** в двоичное, восьмеричное и шестнадцатеричное.

Д.18. Укажите двоичное представление десятичного числа **417**. Затем найдите дополнения **417** до единицы и до двух.

Д.19. Какой результат получится, если дополнение числа до единицы сложить с самим числом?

Ответы на упражнения для самоконтроля

Д.1. **10, 2, 8, 16.**

Д.2. Меньше.

Д.3. Неверно.

Д.4. Шестнадцатеричное.

Д.5. Неверно — наибольшая цифра в любой системе счисления на единицу меньше основания.

Д.6. Неверно — наименьшей цифрой в любой системе счисления является ноль.

Д.7. 1 (основание в нулевой степени).

Д.8. Основанию системы счисления.

Д.9. Заполните пропуски в таблице позиционных значений для данных систем счисления:

десятичная	1000	100	10	1
шестнадцатеричная	4096	256	16	1
двоичная	8	4	2	1
восьмеричная	512	64	8	1

Д.10. Восьмеричное **6530**; шестнадцатеричное **D56**.

Д.11. Двоичное **1111 1010 1100 1110**.

Д.12. Двоичное **111 011 001 110**.

Д.13. Двоичное **0 100 111 111 101 100**; восьмеричное **47754**.

Д.14. Десятичное **2+4+8+32+64=110**.

Д.15. Десятичное **7+1*8+3*64=7+8+192=207**.

Д.16. Десятичное **4+13*16+15*256+14*4096=61396**.

Д.17. Перевод десятичного числа **177**

в двоичное:

```
256 128 64 32 16 8 4 2 1
128 64 32 16 8 4 2 1
(1*128)+(0*64)+(1*32)+(1*16)+(0*8)+(0*4)+(0*2)+(1*1)
10110001
```

в восьмеричное:

```
512 64 8 1
64  8  1
(2*64)+(6*8)+(1*1)
261
```

в шестнадцатеричное:

```
256 16 1
16 1
(11*16)+(1*1)
(B*16)+(1*1)
B1
```

Д.18. Двоичное:

```
512 256 128 64 32 16 8 4 2 1
256 128 64 32 16 8 4 2 1
(1*256)+(1*128)+(0*64)+(1*32)+(0*16)+(0*8)+(0*4)+(0*2)+(1*1)
110100001
```

Дополнение до единицы: 001011110

Дополнение до двух: 001011111

Проверка: исходное двоичное число + его дополнение до двух

```
110100001
001011111
---------
000000000
```

Д.19. Ноль.

Упражнения

Д.20. Некоторые утверждают, что наши вычисления были бы проще в системе счисления с основанием **12**, поскольку **12** делится без остатка на большее количество чисел, чем **10** (для основания **10**). Какой будет наименьшая цифра в системе счисления с основанием **12**? Что может служить символом для наибольшей цифры в данной системе счисления? Чему при этом равны позиционные значения для четырех крайних правых позиций?

Д.21. Как наибольшая цифра в обсуждавшихся выше системах счисления соотносится с позиционным значением первой цифры слева от крайней правой цифры для любого числа в данных системах счисления?

Д.22. Заполните пропущенные позиции в следующей таблице позиционных значений для каждой из указанных систем счисления:

десятичная	1000	100	10	1
по основанию 6	...	...	6	...
по основанию 13	...	169	...	...
по основанию 3	27	...	...	...

Д.23. Преобразуйте двоичное число **100101111010** в восьмеричное и в шестнадцатеричное.

Д.24. Преобразуйте шестнадцатеричное число **3A7D** в двоичное.

Д.25. Преобразуйте шестнадцатеричное число **765F** в восьмеричное. (Подсказка: сначала преобразуйте **765F** в двоичное число и затем это двоичное число — в восьмеричное.)

Д.26. Преобразуйте двоичное число **1011110** в десятичное.

Д.27. Преобразуйте восьмеричное число **426** в десятичное.

Д.28. Преобразуйте шестнадцатеричное число **FFFF** в десятичное.

Д.29. Преобразуйте десятичное число **299** в двоичное, восьмеричное и шестнадцатеричное представления.

Д.30. Приведите двоичное представление десятичного числа **779**. Затем напишите дополнения **779** до единицы и до двух.

Д.31. Каков будет результат сложения дополнения числа до двух и самого числа?

Д.32. Приведите дополнение до двух целого числа со значением **-1** для машины с 32-битным представлением целых чисел.